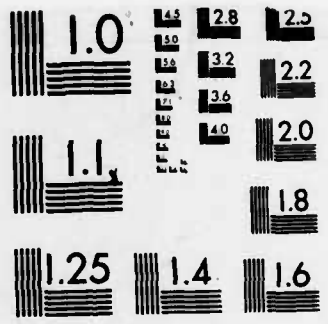
AD-A142 239 RUNTIME DETECTION AND DESCRIPTION OF DEADNESS ERRORS IN ADA TASKING(U) STANFORD UNIV CA COMPUTER SYSTEMS LAB D HELMBOLD ET AL. NOV 83 TR-83-249 AFOSR-TR-84-0464 1/ 1

UNCLASSIFIED N00039-82-C-0250 F/G 9/2 NL

END
DATE FILMED
7-84
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

COMPUTER SYSTEMS LABORATORY

DEPARTMENTS OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
STANFORD UNIVERSITY · STANFORD, CA 94305-2192

Runtime Detection and Description of Deadness Errors in Ada Tasking

D. Helmbold and D.C. Luckham

Program Analysis and Verification Group Report No. 22

Technical Report No. 83-249

November 1983

This research was supported by the Air Force Office of Scientific Research under Contract AFOSR 83-0355 and Advanced Research Projects Agency of the Department of Defense under Contract NOO-039-82-C-0250.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFOSR-TR- 84-0464
6a. NAME OF PERFORMING ORGANIZATION Stanford University 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Air Force Office of Scientific Research
6c. ADDRESS (City, State and ZIP Code) Depts of Electrical Engineering and Computer Science Stanford, CA 94305-2192	7b. ADDRESS (City, State and ZIP Code) Directorate of Mathematical & Information Sciences, Bolling AFB DC 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFOSR 8b. OFFICE SYMBOL (If applicable) NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER AFOSR-83-0355
8c. ADDRESS (City, State and ZIP Code) Bolling AFB DC 20332	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. 61102F; PROJECT NO. 2304; TASK NO. A2; WORK UNIT NO.
11. TITLE (Include Security Classification) Runtime Detection and Description of Deadness Errors in Ada Tasking	
12. PERSONAL AUTHOR(S) D. Helmbold and D.C. Luckham	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) Nov 1983 15. PAGE COUNT
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB GR.	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

A runtime monitoring system for detecting and describing tasking errors in Ada programs is presented. Basic concepts for classifying tasking errors, called deadness errors, are defined. These concepts indicate which aspects of an Ada computation must be monitored in order to detect deadness errors resulting from attempts to rendezvous or terminate. They also provide a basis for the definition and proof of correct detection. Descriptions of deadness errors are given in terms of the basic concepts.
The monitoring system has two parts: (1) a separately compiled runtime monitor that is added to any Ada source text to be monitored, and (2) a pre-processor that transforms the Ada source text so that necessary descriptive data is communicated to the monitor at runtime. Some basic preprocessing transformations and an abstract monitoring for a limited class of errors were previously presented in (2) . Here an Ada implementation of a monitor and a more extensive set of pre-processing transformations are described. This system provides an experimental automated tool for detecting (cont'd on back)

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Dr. Robert N. Buchal	22b TELEPHONE NUMBER (Include Area Code) (202) 767- 4939 22c OFFICE SYMBOL NM

DD FORM 1473, 83 APR ... IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE

Runtime Detection and Description of Deadness Errors in Ada Tasking

D. Helmbold and D.C. Luckham

Program Analysis & Verification Report No. 22

Technical Report No. 83-249

November 1983

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract

A runtime monitoring system for detecting and describing tasking errors in Ada programs is presented. Basic concepts for classifying tasking errors, called deadness errors, are defined. These concepts indicate which aspects of an Ada computation must be monitored in order to detect deadness errors resulting from attempts to rendezvous or terminate. They also provide a basis for the definition and proof of correct detection. Descriptions of deadness errors are given in terms of the basic concepts.

The monitoring system has two parts: (1) a separately compiled runtime monitor that is added to any Ada source text to be monitored, and (2) a pre-processor that transforms the Ada source text so that necessary descriptive data is communicated to the monitor at runtime. Some basic preprocessing transformations and an abstract monitoring for a limited class of errors were previously presented in [2]. Here an Ada implementation of a monitor and a more extensive set of pre-processing transformations are described. This system provides an experimental automated tool for detecting deadness errors in Ada83 tasking and supplies useful diagnostics. The use of the runtime monitor for debugging and for programming evasive actions to avoid imminent errors is described and examples of experiments are given.

Key Words and Phrases: concurrent programming, Ada, debugging aids, error handling and recovery, deadlocks, Ada tasking, program reliability, multiprocessing/multi-tasking, fault-tolerance, modeling and prediction.

1. INTRODUCTION

Errors caused by failure in communication between parallel threads of control in a computational system are called *deadness errors*. As a consequence of such failures, certain threads of control (or sometimes all threads in an entire system) cannot proceed with their computations and hence become "dead". Deadness errors in general occur unpredictably. Whether or not a possible deadness error in a system will occur during system operation may depend on a multitude of external factors, e.g. compilation techniques, run-time scheduling, I/O processing times and external interrupts. They are often extremely difficult to reproduce and locate using current testing methods.

Deadness errors have been described in the past by concepts such as *deadlock*, *blocking*, and *starvation*. These early concepts provided meaningful classification of certain kinds of errors that could occur in 1960's vintage parallel (or pseudo parallel) systems such as simple operating systems. However they are too vague for describing the kinds of deadness error that can occur in a parallel system implemented using the multi-tasking facilities of Ada. For example, problems involving dependent tasks may prevent a Master from terminating [Ada 83, section 9.4]. Such errors could sometimes be described either as deadlock or blocking, but either terminology is essentially inaccurate. The need to develop new descriptive terminology becomes even more obvious in systems using dynamic activation of tasks. The description must not only indicate the cause of the error but must also relate the dynamically generated names of the tasks involved with the origin of those tasks in the source text. Before we can expect to develop an ability to deal with deadness in future parallel systems, we must first provide adequate methods of classification and description.

When dealing with deadness in Ada or other languages of similar complexity, it is useful to divide the problem into three sub-problems: *(1) detection*, *(2) description*, and *(3) avoidance*. Detection involves recognizing a dead state, and usually requires less information than description. Description involves providing sufficient information to locate the source of an error in Ada text. Avoidance involves both style guidelines for constructing error-free systems, and programming techniques for evasion of imminent errors at run-time.

In this paper we investigate the application of run-time monitoring methods to these three sub-problems. Alternative methods of eliminating deadness errors based on static analysis at compile time are not addressed in this paper. So far, the known static analysis methods are very difficult and time-consuming in the general case [5].

In Chapter 2, concepts for classifying deadness errors in Ada tasking are defined. These concepts are derived from the informal semantics of Ada tasking given in [1]. They form a complete set in the sense that an operational description of Ada tasking can be given using only these concepts. Our monitor implementation is based on these concepts. However, we feel that our present set of concepts should be treated as tentative. It is possible to define other complete sets of concepts. Alternative concepts with advantages over the present set may emerge as experience in this area accumulates.

Our monitor system has two parts: *(1)* a separately compiled run-time monitor written in Ada, and *(2)* a preprocessor that transforms Ada source text so that necessary descriptive data is communicated to the monitor at run-time. The result of applying the preprocessor to any legal Ada program is a modified program which is again a legal Ada program and contains the monitor. When this modified program is run, sufficient information about tasking activities in the original program will be passed to

the monitor, enabling it to detect imminent dead states and provide descriptive information. The transformations currently implemented in our present preprocessor extend the set of transformations previously given in [2] in two ways: *(1)* the set of deadness errors detected by the monitor is extended to include errors involving the inability to terminate, *(2)* the monitored data is extended to include data necessary to give an adequate description of a deadness error for the purpose of debugging and evasive action. Also the previous paper lacked discussion of many important implementation details upon which the correctness of an actual implementation depends.

The present monitor has a number of deficiencies. It does not work correctly on programs that use task abortion or priorities or execute tasking statements during elaboration of declarative parts. It will not detect deadness errors due to task communication by means other than rendezvous (e.g. by shared variables). The implementation is described here in sufficient detail to indicate how run-time monitoring techniques can be extended beyond the capabilities of our present monitor to detect and diagnose a wider class of deadness errors.

An Ada implementation of the run-time monitoring system is described in Chapter 3. This description encompasses *(1)* the descriptive data about tasking states that is monitored, *(2)* representation of the descriptions and processing to detect errors, and *(3)* structural design of the monitor. The monitored data must be sufficient both for detection of deadness and for providing diagnostics. The actual monitor data structures and procedures must correctly implement representations of scheduling states (as defined in Chapter 2); any monitor procedure must always terminate, preferably as quickly as possible. The actual design (structure) of the monitor is an important consideration both for run-time efficiency and to reduce recompilation if the monitor system is altered for a special application. The design of the present monitor is simple and conservative to ensure correctness; more efficient distributed designs are currently being developed.

Chapter 4 describes the preprocessing transformations applied to Ada source text. The description deals with the complete set of transformations that are currently implemented. The details are complex; our description is therefore presented informally and relies on illustrative examples. The preprocessor is implemented in SNOBOL; reimplementation in Ada is planned.

The monitoring system may be used not only for recognition of errors but also for *evasive action* programming. Essentially, the monitor "knows" a deadness error is certain to happen (if the computation continues normally) before it occurs. Warnings (e.g. Ada exceptions) may therefore be propagated to the monitored program before the error occurs, thus enabling it to evade the error by taking some abnormal course of action. Such evasion may be temporary in that the error may become imminent again, but the program can continue useful operation for a time. It may then have to evade again, and so on. These evasive action techniques need to be investigated and developed since they may be a very useful method of keeping large multi-tasking systems in operation in the presence of deadness errors. Eventually one would hope to be able to determine at compile time that such systems are free of deadness errors, but until the necessary theory of static detection is developed, evasive action may become just as important a way of dealing with deadness errors as testing methods are for most other kinds of errors today. Indeed, if a system has to deal with unreliable elements, as happens in many practical applications, proofs of freedom from deadness cannot be given and evasive action techniques based on run-time monitoring could become a standard programming practice.

Some techniques for evasive action programming are given in Chapter 5. These are very modest and represent just a beginning. Examples of monitoring experiments for debugging and evasive action are given in Chapter 6.

The current experimental monitor is programmed in Ada and compiled using the Adam compiler at Stanford [4]. Since Adam does not support all of Ada83, some parts of the monitor implementation have used circuitous techniques. This is especially evident in our implementation of evasive action; warnings are implemented by means of extra parameters of the monitor entries instead of exceptions because Adam does not support exception propagation during task rendezvous.

Our run-time monitor implemented in Ada is an independent source level tool. One can argue that the monitor should be a part of the underlying run-time supervisor. Incorporating the monitor into the supervisor has several advantages, including: *(1)* Most of the preprocessing can be omitted, since the monitor can make use of calls to the supervisor inserted by the compiler, and the supervisor's representation of task IDs. *(2)* The monitor's representation of the state of task interactions will be more accurate since it is able to observe the program's actual scheduling state (however this may not be true in the case of a supervisor distributed over a multiple CPU system). *(3)* The monitor and the supervisor can share a single data structure, rather than maintaining two copies of almost identical data. Conversely, separating the monitor from the run-time supervisor also has advantages. *(1)* It allows us to focus on deadness monitoring independent of specific supervisors and scheduling algorithms. Since both run-time monitors and supervisors depend on currently active research areas (deadness error detection and description techniques on one hand, and implementation of Ada tasking semantics on the other), the divide and conquer approach of separating out the monitor makes the development of both tools easier. If the run-time supervisor is also implemented in Ada, it should be relatively easy to integrate the two packages at an appropriate time in the future. *(2)* Portability: a separate run-time monitor in Ada is completely portable, being processable by any compiler and executable in any Ada environment; a monitoring system integrated into a particular supervisor will almost certainly be dependent on the underlying machine and implementation of tasking. *(3)* Source level monitoring could be particularly advantageous in integrating compile-time and run-time deadness error detection techniques as advocated in [6] where the choice of monitor may depend on features of the source program.

2. DEFINITIONS

2. DEFINITIONS

This chapter presents a set of concepts that are the basis for defining deadness errors and implementing the monitoring system. These concepts are also used to define a notion of "correct monitoring".

2.1 TASK STATUSES

According to the semantics of tasking [1] a task may be in any one of the following statuses; a status has information associated with it:

1. **Running:** a task in this status may be run. This is the only status in which a task may run.

2. **Calling:** task t has issued an entry call, s.e, to task s, which is neither conditional nor timed. The task s and the entry e are associated with the Calling status of t.

3. **Accepting:** a task t is waiting for an entry call at an accept statement or at a selective wait statement that does not have an else clause, open terminate alternative, or an open delay alternative. The set of entries being waited for (i.e., the entry of the accept or those entries corresponding to open accept alternatives of the select) is associated with the Accepting status of t.

4. **Select_Terminate:** a task t is at a selective wait statement with an open terminate alternative; the set of entries corresponding to open accept alternatives and the set of tasks dependent on t are associated with the Select_Terminate status of t.

5. **Select_Dependents_Completed:** task t is at a selective wait statement with an open terminate alternative and all dependent tasks have reached either Terminated status or Select_Dependents_Completed status. The set of entries corresponding to open alternatives of the select statement is associated with this status.

6. **Block_Waiting:** task t has reached the end of an inner block or subprogram and is waiting for the tasks dependent on the inner scope to terminate; the set of tasks dependent on the block or subprogram is associated with the Block_Waiting status of t.

7. **Completed:** task t has completed. The set of tasks dependent on t is associated with the Completed status of t.

8. **Terminated:** task t is terminated. No additional information is associated with this status.

Notes:
A task executing a delay or else part of a selective wait statement is considered to be in status Running

Blocked: A task in any of the statuses 2 - 8 is said to be *blocked*.

Finished: A task is *finished* if it has status Terminated or Select_Dependents_Completed. (*Note:* as a consequence, if a task is finished then all its dependents are finished.)

This set of statuses and associated information is sufficient to describe that part of the Ada semantics of task rendezvous that determines the schedulability of a task. Such a description may be given by means of a status change diagram indicating how the semantics of rendezvous determines the status changes of a task. Some status changes of task t are *direct* in the sense that the action of t itself causes the change. Other status changes of t are *indirect* in the sense that they are a consequence of the state of the tasking system and are not caused by an action of t itself.

Direct Status Changes:

```
Running → Calling                -- Simple entry call issued.
Running → Accepting              -- Accept statement or set of accept
                                 -- alternatives reached.
Running → Select_Terminate       -- Selective wait with open terminate
                                 -- alternative reached.
Running → Block_Waiting          -- End of inner block or subprogram reached.
Running → Completed              -- End of task body reached.
```

Indirect Changes:

```
Calling                          → Running      -- Rendezvous completed.
Running                          → Calling      -- Conditional or timed entry call
                                                -- accepted.
Accepting                        → Running      -- Open entry is called.
Select_Terminate                 → Running      -- Open entry is called.
Select_Terminate                 → Select_Dependents_Completed
                                                -- All dependents of task finish.
Select_Dependents_Completed      → Terminated   -- Master terminates.
Select_Dependents_Completed      → Running      -- Open entry is called.
Block_Waiting                    → Running      -- All dependents of block finish.
Completed                        → Terminated   -- All dependents of task finish.
```

Notes:

A task executing a delay statement is in status Running. The indirect status change from Accepting to Running occurs when the entry call is issued rather than when the rendezvous is initiated. A task changes status from Running to Calling after having issued a conditional or timed entry call only if the call is accepted (this status change is therefore indirect). A task which executes a selective wait statement will usually change from Running to Accepting. A task which executes the else part (or delay alternative) of a select statement remains in status Running.

Our indirect status change algorithm for the terminate alternative differs from Ada83. The two algorithms are equivalent. When a subtree of finished tasks can be terminated, our status changes terminate from the top down. The Ada LRM terminates tasks from bottom up. In both cases, the whole subtree can be terminated (see Section 3.3).

2.2 SCHEDULING STATES AND DEADNESS ERRORS

For a given input, a program P may have many different possible computations. Each possible computation is the result of a legal Ada scheduling of the runnable tasks. Here, the word "scheduling" is used in a very broad sense to reflect simply the order in which changes of status occur among the individual tasks of P. Different orders may result from different scheduling algorithms for multiplexing tasks on a single CPU, or from differing speeds of CPU's in a multiprocessor system. The details of the underlying scheduling do not concern us in this paper. We are concerned only with observable differences in the sequence of status changes. It should be noted that different schedulings may result in different outputs from the computation, e.g. in the case where P is monitoring its own status changes.

Task Identifiers. Each task that is activated during a computation of a program is assigned a unique name called its *identifier*. It is assumed that a task can access its own identifier and the identifier of any task that is visible to it.

Execution. An *execution* of P is a sequence of pairs consisting of a task identifier and a simple statement such that:

1. the task identifier of the first pair identifies the main program;
2. the task identifier of the nth pair $\langle t_n, c_n \rangle$ has status Running after the execution of the statements in the previous pairs by the named threads of control;
3. as a consequence of the completed execution of the statements in the previous pairs in the sequence by the named threads of control, t_n may legally complete execution of the simple statement, c_n.

Executions correspond to computations of P on a single CPU. An execution can be constructed from an actual computation. When a simple statement completes normally, a pair consisting of the identifier of the executing thread of control followed by the simple statement is added to the execution sequence. Conversely any execution corresponds to an actual computation on a single CPU under some scheduling. Since the semantics of Ada are independent of the number of CPU's, definitions based on this imposed linearization of tasking computations are equivalent to computations under any scheduling.

Notes:
It is convenient to consider **begin** and **end** as simple statements in the definition of execution.
Statements appear in executions in positions corresponding to their completion (i.e., normal termination). Completion of a subprogram call follows completion of the subprogram body. For example, if task t calls procedure p, then the simple statements executed during p's execution will appear in an execution pair for t *before* the procedure call appears. An entry call completes when the calling task is placed on the corresponding entry queue or the call is accepted; the calling task does not return to status Running until completion of the rendezvous. If a task t makes an entry call, s.e, then the pair <t, s.e> will appear before any pairs containing statements in an appropriate accept body, and pairs representing completion of an accept body must appear before any further pair containing t.
The concept of execution described here can be given a formal definition in terms of transition rules similar to the operational semantics for Ada in [3]. We may therefore use the notions "computation" and "execution" interchangeably in the following discussion.

Scheduling. A *scheduling* is an activity which may change the execution sequence of P given a fixed input.

Task-Status Pairs. A *task-status pair* is an ordered pair consisting of a task identifier as the first element and a status as the second element (notation: <t, s>).

Scheduling State. A *scheduling state* is a set of task-status pairs such that no two pairs have the same task identifier.

A scheduling state is associated with every position in an execution of P. A scheduling state at a point is a set of task-status pairs such that each task activated up to that point in the execution is the first element of exactly one pair and has its status as the second element. If <t,s> is a member of state S, then task t has status s in S.

Sequences of Scheduling States. A computation of program P has an associated linear sequence of scheduling states. Each new state in the sequence results from the previous state by a status change by (or activation of) one task. Simultaneous status changes are ordered arbitrarily; an indirect status change follows the status change of the task causing it. All tasks are activated in Running status.

Deadness Error. A *deadness error* is a scheduling state occurring in a computation of P in which some task t is blocked but not terminated, and there can be no possible continuation of that computation of P in which the status of t has changed. When such an error occurs, task t is said to be *dead*.

Potential Deadness Error. Program P has a *potential deadness error* if there is an input and a possible computation of P such that the associated sequence of scheduling states contains a deadness error.

Some deadness errors can be described as follows:

Global Blocking is a scheduling state in which no task has status Running, no (indirect) status changes are possible, and not every task has status Terminated.

Circular Deadlock. A *circular deadlock* is a deadness error in which a subset of tasks are all in status Calling and the calls are to entries of members of the subset.

Example 1: Deadness involving inability to terminate.

```
...
task T1 is
    entry E1;
end T1;

task body T1 is
    task T2;
    task body T2 is
    begin
        T1.E1;
    end T2;
begin
    null;
end T1;
...
```

A dead state will occur in which T2 has status Calling (T1.E) and T1 has status Completed. These statuses can never change. Since T2 is dependent on T1, T1 cannot terminate; and T2 can never leave Calling status. Our monitor will detect this error.

Example 2: Deadness occurring during elaboration.

```
declare
    task T1 is
        entry E1;
    end T1;
```

```
    function F return INTEGER is
    begin
        T1.E1;
        ...
    end F;

    X : INTEGER := F;
    ...
begin
    ...
```

Initialization of X requires the completion of an entry call to T1; T1 will not be activated until the elaboration of the declarative part is completed. This elaboration leads to a dead state in which the elaborating task has status Calling (T1.E1). Our monitor will *not* detect this error.

Notes:
Deadness does not include many situations commonly referred to as starvation which result from the underlying scheduling (in the broad sense used above). Whether or not a task is dead will often depend on properties of the program P. Our run-time monitoring techniques detect dead states that can be recognized using only the syntax and tasking semantics of the programming language, Ada.

2.3 MONITORED PROGRAMS

Run-time monitoring for deadness errors involves modifying a given program P and adding a monitoring system M. The program P is modified so that any activated task will have a unique identifier, and tasks may identify each other and communicate status changes to M. The resulting program, P', is called a *monitored* program. It is important to establish that the original program P and the transformed program P' have the "same" set of potential deadness errors in some sense. The next set of definitions establish when P and P' can be said to possess the same potential deadness errors. These definitions are very general because they must take account of the possible dynamic creation of tasks in Ada and corresponding dynamic allocation of task identifiers.

Correspondence: We assume there is a textual correspondence between P and P' such that:

1. every declarative region in P corresponds to a declarative region in P',
2. every declaration in P of a type or program unit (in the Ada sense) corresponds to a declaration in P' of the same kind,
3. every object in P corresponds to an object or component object in P' of the same kind,
4. every statement in P corresponds to a statement in P' of the same kind,
5. declarations, objects, and statements in a region R in P correspond to declarations, objects, and statements in the corresponding region R' in P'.

Notes:
Any object declared in P corresponds to an object (or component) declared in P' of the same kind, In particular tasks correspond to tasks. However, not every declaration or statement in P' need have a correspondence in P.

Corresponding Executions. Let E and E' be executions of P and P' respectively. Assume there is a textual correspondence between P and P'. Then E and E' *correspond* if all task-statement pairs of E

can be placed in a correspondence with task-statement pairs in E' according to the following inductive test: Suppose that the first n pairs of E correspond to pairs (in the same order) among the first m pairs of E' ($m \geq n$), and that there is a one — one correspondence between all the task identifiers that have occurred so far in E and a subset of those in E'. Let the nth. and mth. pairs be $\langle t_n, c_n \rangle$ in E and $\langle t_m, c_m \rangle$ in E'.

1. if statements c_m and c_n are in correspondence (under the textual correspondence between P and P'), then both the pairs and the tasks must correspond. If t_n and t_m already correspond, then the pairs are said to correspond and the test proceeds to the next pairs in E and E'. If neither task yet corresponds to a task, the pairs and the tasks are placed in correspondence and the test proceeds to the next pairs in E and E';
2. if c_m does not correspond to any statement in P then $\langle t_n, c_n \rangle$ is compared with the next pair in E';
3. if neither of the first two cases holds, then the correspondence test fails.

Notes:
If two executions E and E' correspond then the task identifiers in E are in one-one correspondence with a *subset* of the task identifiers in E'. If t in E corresponds with t' in E' then t executes code corresponding to some of the code executed by t', possibly interspersed with code in E' which has no correspondence in E. Thus, in a general sense corresponding task identifiers are names for threads of control that execute the same subcomputations (restricted to statements of P). E' may have tasks that do not correspond to any task in E; this is a consequence of the assumption that the textual correspondence between P and P' is "into", i.e., P' may be "bigger" than P.

Equivalent Scheduling States. If E and E' are corresponding executions of P and P' then scheduling state S of E is **equivalent** to a scheduling state S' of E' if for every task-status pair $\langle t, s \rangle$ in S the task-status pair $\langle t', s \rangle$ is in S' where t and t' correspond in E and E', and all other tasks of S' are blocked.

Equivalent Potential Errors. P and P' have equivalent potential deadness errors if for every potential deadness error of P occurring in execution E, say, there is a corresponding execution E' of P' in which an equivalent deadness error occurs, and conversely.

Note:
"Conversely" means the following: if a deadness error S' occurs in execution E' of P' then there is an execution E of P such that E and E' correspond and a deadness error S equivalent to S' occurs in E.

Correct Monitoring: Correctness is taken to mean: *(1)* for any potential deadness error of the original program P there is an equivalent potential deadness error in the monitored program P' and conversely, *(2)* in any computation of P', if the monitor detects a deadness error, it will do so *before* that error occurs and that error will occur if the computation continues normally, *(3)* certain kinds of deadness errors, including global blocking and circular deadlock will always be detected.

Notes:
(1) means that addition of the monitor does not change the set of potential deadness errors of the monitored program. *(2)* does not imply that the monitor will detect every deadness error, as defined in Section 2.2, but that any error it does detect will be a future scheduling state of P'. *(3)* is a completeness requirement.

A formal treatment of correctness with detailed proofs is beyond the scope of this paper. Proof of

correct monitoring can be based on properties of the monitor implementation (Chapter 3) and the preprocessing (Chapter 4). The monitor implementation ensures that *(1)* all monitor entry calls terminate; *(2)* the monitor correctly represents the scheduling state implied by any legal sequence of monitor entry calls; *(3)* the monitor will detect any instances of global blocking or circular deadlock arising in its representation. The preprocessing transforms P into P' such that *(1)* there is a textual correspondence between P and P', *(2)* the monitor will be able to predict the occurrence of deadness errors in P' correctly (Section 3.5), and *(3)* P and P' have equivalent potential deadness errors.

3. DEADNESS MONITOR.

The monitor detects deadness errors and provides diagnostic descriptions based on information received from the preprocessed program. In our implementation this information consists of changes of statuses and associated information (see Chapter 2). The monitor maintains, throughout the execution of the modified program, a "picture" of the program's scheduling state. This picture is updated and checked for deadness errors when information is received from the program. In addition to detection and diagnostics, the monitor also provides facilities for tracing status changes, querying the current "picture" and undertaking evasive action to avoid a deadness error.

3.1 THE MONITOR STRUCTURE

The monitor is implemented in two units, a task and a package. The task is inserted into the program by the preprocessor. The package is designed to be compiled separately; it contains the monitor's data structure and the procedures that act upon it. It is compiled only once, and then linked to each program to be monitored. The monitor task's main purpose is to protect the monitor package. The preprocessed program communicates status change information to the task by means of entry call parameters. The monitor task then calls the appropriate procedure of the monitor package. Buffering the information through a task in this way ensures that only one thread of control (the monitor task) can update the monitor's data structure at a time. The monitor task also provides a convenient place to encapsulate facilities that may need to be modified for specific applications; e.g. an interactive version of the monitor has been implemented by modifying only the monitor task.

Outline of the Monitor Structure:

```
-- Separately compiled package:

    package MONITOR_DATA_PACKAGE is
        ...                                 -- Data structures for the monitor's picture.

        procedure INIT;
        procedure A;                        -- Subprograms for updating the
        procedure B;                        -- monitor's picture.
            ...
    end MONITOR_DATA_PACKAGE;

-- Preprocessed Ada Program:

    with MONITOR_DATA_PACKAGE;
        ...                                 -- Outermost declarative part.
    task MONITOR is
        entry A;                            -- Entries match monitor data package
        entry B;                            -- procedures.
        ...
    end MONITOR;
        ...                                 -- Declarations of program to be monitored.
```

```
task body MONITOR is

begin
    MONITOR_DATA_PACKAGE.INIT;
    while not MONITOR_DATA_PACKAGE.DONE loop
        select
            accept A do
                MONITOR_DATA_PACKAGE.A;
            end accept;
        or
            accept B do
                MONITOR_DATA_PACKAGE.B;
            end accept;
        or
            ...
        end select;
    end loop;
end MONITOR;
...                                    -- Bodies of units in monitored program.
```

Note:
All rendezvous with the monitor task are assumed to terminate and not to nest (i.e., contain) other rendezvous.

3.2 THE MONITOR PICTURE

The monitor maintains, at run-time, a picture of the program's scheduling state. The picture is in the body of the monitor data package. This picture consists of: status and associated information for each task, lengths of entry queues, task dependencies, and several global (to the monitor package) counters. This picture is incomplete in that it does not reflect any interactions with the monitor task itself. More important, at some points, this picture may not correspond exactly with the actual scheduling state of the monitored program (see Section 3.5 for a discussion of how this can occur, and why it is not critical).

3.2.1 TASK INFORMATION

Each activated task of the monitored program (except the monitor itself) is represented by a record in the monitor's data structure. This record contains status and other information pertaining to the task.

```
type TASK_STATUS_RECORD is                -- Each task will have a record of
                                          -- this type to hold information
                                          -- associated with the task.
    record
        TASK_NAME     : NAME_STRING;      -- The user-defined source text name.
        STATUS        : TASK_STATUS;      -- The status of this task.
        CALLED_TASK   : TASK_ID;          -- The task that this task has issued an
                                          -- entry call to.
        CALLED_ENTRY  : NAME_STRING;      -- The entry being called.
        PARENT_TASK   : TASK_ID;          -- The task that this one depends on.
        DEPENDENTS    : ID_PTR;           -- A list of tasks depending on this task.
```

```
        NUM_WAIT_FOR : INTEGER;        -- The number of tasks that need to finish
                                       -- before this one can proceed.
        LIST_PTR     : ENTRY_LIST;     -- A pointer to the list of entries in this task
        TRACE        : BOOLEAN;        -- True IFF trace information
                                       -- on this task is to be printed
    end record;
```

The first component contains the task name. This string is used only to relate the task to its declaration in the Ada source text of the monitored program; it is not used in detecting errors. The second component contains the task's status (see Section 2.1). The next two components contain associated information for status Calling: the task and entry called. Following these are components containing dependency information: a list of dependent tasks that this task is waiting on; the number of those tasks that have not terminated; and this task's parent (see Section 3.2.3). An additional component holds a pointer to the list of entries of the task. The last component contains a flag indicating whether or not the task's status changes should be traced. These records are stored in an array in the monitor data package body and indexed by task IDs.

Note:
Some task status record components will be irrelevant, e.g. if a task has status Running then the CALLED_TASK and CALLED_ENTRY components are irrelevant.

3.2.2 ENTRY INFORMATION

The monitor creates an entry record for each entry of a task just before that entry is first referenced at a call, accept or select statement. These records contain the unique string name for the entry (created by the preprocessor, see 4.4), the number of tasks calling the entry, and a HERE_FLAG, indicating if the task is currently waiting for (ready to accept) a call to the entry. All of the records for a task's entries are stored in an unordered linked list referenced from the task's status record.

```
type ENTRY_DATA_RECORD;
type ENTRY_LIST is access ENTRY_DATA_RECORD;
type ENTRY_DATA_RECORD is
    record
        NAME       : NAME_STRING; -- Unique string identifier.
        QUEUE_SIZE : INTEGER      -- Number of tasks calling.
        HERE_FLAG  : BOOLEAN      -- Waiting at entry.
        NEXT       : ENTRY_LIST;  -- Rest of the entries.
    end record;
```

3.2.3 DEPENDENCY LISTS

Keeping track of dependencies poses special problems for the monitor implementation. If the monitor were to hold all task dependency information, then it would have to maintain stacks of masters for each task, and masters would have to be assigned unique IDs. To avoid this, lists of dependent tasks are maintained in the monitored program itself. In each potential master of the original program, the preprocessor inserts a list containing all of the tasks directly dependent on that master (Section 4.1). The preprocessor inserts an additional list in each task body (and main program) containing all the *sons* of that task (Section 3.4). These dependency lists can only be operated on by monitor procedures and are thereby protected from simultaneous access. Whenever the monitor is required to have access to a dependency list (e.g. if that list contains information associated with the current status of a task — Section 2.1) it is passed a pointer to that list.

3.2.4 GLOBAL BLOCKING

Three variables are used to enable the monitor to efficiently detect global blocking. The monitor maintains counts of:

1. the number of tasks that have been activated, NUM_TASKS;
2. the number that are blocked, NUM_BLOCKED; and
3. the number that have terminated, NUM_TERMINATED.

If the number of tasks that are terminated is equal to the number of tasks that have been activated then the program has terminated. Otherwise, if the number of tasks that are blocked and terminated is equal to the number of tasks that have been activated, then global blocking has occurred. These checks are done every time a task becomes blocked (for any reason) in the monitor's picture.

An additional boolean variable, DONE, is used to inform the monitor task that all of the other tasks have terminated. This variable is declared in the visible part of the monitor package so it can be examined by the monitor task.

3.3 TASK TERMINATION IN THE MONITOR'S PICTURE

The monitor must be able to distinguish between a global blocking situation and a program's normal completion. This requires that the monitor recognize when tasks may be terminated. The monitor's algorithm for changing a task's status to Terminated is complex, involving several different monitor entries. This section describes the algorithm in its entirety. The contribution of each monitor entry is described in Section 3.4.

We define the sons of task t (or the main program) to be those tasks which:

1. directly depend on t;
2. directly depend on one of t's inner blocks; or
3. directly depend on a subprogram (or subprogram inner block) elaborated by t.

If task s is the *son* of task t, then task t is the *parent* of task s. This parent-son relationship forms a tree structure. All tasks dependent on t will be located in the subtree rooted at t.

If task t has *finished* (section 2.1), then so have all the sons of t. Thus, by induction, all tasks in the subtree rooted at t have finished.

When task t is ready to complete, it passes the list of all its sons to the monitor and then reaches Completed status. The monitor sets the PARENT_TASK component of the task status record for each task on the list to t's ID. The monitor stores the number of sons that have not yet finished in t's NUM_WAIT_FOR component. As the sons of t finish, the NUM_WAIT_FOR count in t's status record will be decremented; thus this component contains the number of t's sons which have not yet finished. By checking to see if task t's NUM_WAIT_FOR component is 0, the monitor can determine if all the sons of t have finished. When this occurs, task t is terminated, along with all of its dependents (direct and indirect) that are at select statements with open terminate alternatives. Since t has now terminated, we may have to decrement the NUM_WAIT_FOR component of t's parent. The monitor checks the PARENT_TASK component of t's status record. If it is non-empty (contains a valid task ID) then the PARENT_TASK's NUM_WAIT_FOR count is decremented.

A similar algorithm is used when a task t is ready to leave an inner block (or subprogram). The list of dependents passed to the monitor will contain only those sons of t dependent on the inner block; only these dependents affect t's NUM_WAIT_FOR count. When the count reaches 0, t is placed back into status Running.

A task, t, reaching a select statement with an open terminate alternative (i.e, status Select_Terminate) cannot terminate until all of its dependents have finished. Using the above algorithm, the monitor changes t's status from Select_Terminate to Select_Dependents_Completed when all of t's dependents have finished. The dependents of t *are* not terminated yet. After t's master terminates, t will be terminated, then t's sons will be terminated, and so on. This order of termination is top-down instead of the bottom-up order specified in the Ada LRM, but since all such terminations *are* done immediately (within a single monitor call), the order does not effect the correctness of the monitors' picture.

Notes:
It is important to set the PARENT_TASK component of a status record only when the parent is waiting on that task. Otherwise, the task may decrement its parent's NUM_WAIT_FOR count before the parent is waiting for it (this could lead to incorrect results if the parent were waiting on an inner block).
It is also important to have the monitor modify the lists of dependents. When a task is attempting to terminate, it passes the monitor a list of its dependents. If some other task creates a new dependent of the first task, then the change in the list of dependents must be communicated to the monitor. The monitor checks for this situation whenever it updates a dependency list. The monitor's mutual exclusion property ensures that two tasks are never simultaneously updating a dependency list.
It is possible for a task to "un-finish." If a task at a select statement with a terminate alternative has already decremented its parent's NUM_WAIT_FOR count, and then it accepts an entry call, the NUM_WAIT_FOR count must be incremented; this action takes place when a son changes status from Select_Dependents_Completed to Running.

3.4 MONITOR PROCEDURES

Calls to the monitor task entries (which simply call the monitor data package procedures) are placed in the original program by either the preprocessor or the programmer. The preprocessor inserts all calls needed to inform the monitor of impending status changes. Diagnostic output from the monitor and evasive action are controlled by monitor calls inserted by the programmer.

Evasive action in this implementation must make use of the DEADLK_FLAG formal parameter of monitor entries. An entry call returns the value TRUE for this parameter if and only if a deadness error is detected in the monitor's picture as a result of the call. For details on evasive action see Chapter 5, the DEADLK_FLAG parameter will be ignored for the remainder of this section.

Below is the visible part of the monitor package and the specification for the monitor task. These specifications define the visible data types used in monitoring tasking activity, and the set of entries (and their parameters) provided to inform the monitor of tasking action.

```
-- Data structures used by the monitor. (Compiled separately from the program to be monitored.)

    with DTTY_IO                              -- Adam I/O package.
    package MONITOR_DATA_PACKAGE is
-- Bounds and data structures used by the monitor.
        MAX_NUM_TASKS : constant INTEGER := 100;
        TASK_LIMIT    : constant INTEGER := (MAX_NUM_TASKS -1);

        subtype TASK_ID is INTEGER range -1 .. TASK_LIMIT;
-- Special ID's used for initialization of task IDs and for tracing.

        ALL_TASKS     : constant TASK_ID := -1;
        NULL_TASK     : constant TASK_ID := -1;

        subtype NAME_STRING is STRING(1 .. STRING_SIZE);
        type ENTRY_REC;                       -- ENTRY_PTRs are used to pass lists of
                                              -- entries to the monitor.
        type ENTRY_PTR is access ENTRY_REC;
        type ENTRY_REC is
            record
                NAME : NAME_STRING;
                NEXT : ENTRY_PTR;
            end record;

        type ID_REC;                          -- Used to pass the monitor lists of task ID's
        type ID_PTR is access ID_REC;
        type ID_REC is
            record
                ID   : TASK_ID;
                NEXT : ID_PTR;
            end record;
        ...

        -- Monitor package procedures are omitted since they correspond one — one with
        -- monitor task entries described below.

        DONE : BOOLEAN := FALSE;
    end MONITOR_DATA_PACKAGE;

-- The DEADLOCK MONITOR TASK. (This is inserted into the program to be monitored.)

    use MONITOR_DATA_PACKAGE;
    task MONITOR is

-- Group 1 entries are called to notify the monitor of status changes that are about
-- to take place, activation of new tasks, and task dependencies.

        entry NEWTASK(TASK_NAME : in NAME_STRING;
                      NEW_ID    : out TASK_ID);
        entry ADD_DEPENDENT(PARENT                : in TASK_ID;
                            SON                   : in TASK_ID);
                            BLOCK_DEPENDENTS_LIST : in out ID_PTR;
                            SON_LIST              : in out ID_PTR);
        entry CALLING(CONSUMER     : in TASK_ID;
                      SERVER       : in TASK_ID;
```

```
                    ENTRY_NAME    : in NAME_STRING;
                    DEADLK_FLAG   : out BOOLEAN);
        entry ACCEPTING(SERVER        : in TASK_ID;
                        ENTRY_NAME    : in NAME_STRING;
                        DEADLK_FLAG   : out BOOLEAN);
        entry SELECTING(SERVER          : in TASK_ID;
                        ENTRY_LIST      : in out ENTRY_PTR;
                        TERMINATE_FLAG  : in BOOLEAN;
                        DEPENDENTS      : in ID_PTR;
                        DEADLK_FLAG     : out BOOLEAN);
        entry START_RENDEZVOUS(CONSUMER    : in TASK_ID;
                               SERVER      : in TASK_ID;
                               ENTRY_NAME  : in NAME_STRING);
        entry END_RENDEZVOUS(CONSUMER : in TASK_ID;
                             SERVER   : in TASK_ID;
                             ENTRY_NAME : in NAME_STRING);
        entry END_BLOCK(CONSUMER      : in TASK_ID;
                        DEPENDENTS    : in ID_PTR;
                        DEADLK_FLAG   : out BOOLEAN);
        entry END_TASK(CONSUMER    : in TASK_ID;
                       DEPENDENTS  : in ID_PTR;
                       DEADLK_FLAG     : out BOOLEAN);

-- Group 2 provides some facilities for tracing statuses and scheduling states.

        entry PRINT;
        entry TRACE(SUBJECT : in TASK_ID;
                    FLAG    : in BOOLEAN);

-- Group 3 is used to facilitate evasive action.

        entry QUERY(SUBJECT     : in TASK_ID;
                    CALLED_TASK, ENTRY_CALLED : out NAME_STRING;
                    WAITING_AT : out ENTRY_PTR);
        entry UNBLOCK(SUBJECT : in TASK_ID);

    end MONITOR;
```

3.4.1 GROUP ONE ENTRIES

Calls to group one monitor entries are placed in the original program by the preprocessor (see Section 4). These calls notify the monitor of impending status changes, and any associated information. Such calls typically involve modifying the monitor's picture.

The **NEWTASK** entry informs the monitor that a task has been created. The monitor creates a new task status record, initializing it with the TASK_NAME and status Running. The remaining components are set to null values. The record is stored in the next available position in the array of task records. The index of its position is returned as the NEW_ID.

Notes:
Task IDs cannot be implemented by access type objects accessing task objects because of the strong typing of Ada. The monitor type declarations would have to be changed (and the monitor

recompiled) for each monitored program P. The task type declarations of P would have to be placed in the most global declarative part; and still the problem of a task being able to find its own name would remain.

The **ADD_DEPENDENT** entry is used to put task IDs on dependency lists. When the monitor receives this call, it places SON on the two lists. If either of the LISTs is a part of the PARENT's associated information, then the DEPENDENT's list and the NUM_WAIT_FOR count in the PARENT's status record are updated accordingly.

The **CALLING** entry is used to tell the monitor that a task is about to issue an entry call. When the monitor accepts this entry it undertakes the following actions:

1. change the CONSUMER's status in the monitor's picture from Running to Calling.
2. the task and entry called are stored in the CONSUMER's status record.
3. increment the queue size (in the monitor's picture) associated with the called entry.
4. if, in the monitor's picture, the SERVER is in status Accepting, Select_Terminate, or Select_Dependents_Completed, and it is waiting on the called entry then the SERVER's status is changed to Running and the NUM_BLOCKED count is decremented.
5. the NUM_BLOCKED count is incremented due to the consumer becoming blocked.
6. the picture is checked for circular deadlock involving the CONSUMER.

The **ACCEPTING** entry is used to inform the monitor that a task is about to execute an accept statement. Upon receiving this call the monitor examines the queue-size for this entry. If it is zero, then the SERVER's status is changed to Accepting, the HERE_FLAG for the entry is set, and NUM_BLOCKED is incremented.

SELECTING is called when a task is about to execute a select statement, which may contain a terminate alternative, as well a number of open accept alternatives (see Section 4.4.3). The ENTRY_LIST parameter contains a list of all the entries that can be accepted. The DEPENDENTS parameter holds a list of all the task's sons. The TERMINATE_FLAG parameter will be true only if there is an open terminate alternative. If some of the entries on ENTRY_LIST have non-empty queues (in the monitor's picture), then the SERVER remains in status Running. Otherwise, the HERE_FLAGs for all the entries on the list are set and the TERMINATE_FLAG is checked. If it is true, then:

1. the SERVER is placed in status Select_Terminate.
2. the SERVER's DEPENDENTS component is set to the DEPENDENTS list.
3. if the PARENT_TASK component of the SERVER's status record contains a valid ID, then the PARENT_TASK's NUM_WAIT_FOR count is decremented and checked for 0.

If the TERMINATE_FLAG is false, then the SERVER is put into status Accepting.

If the SERVER is now blocked, NUM_BLOCKED is incremented.

The **START_RENDEZVOUS** entry is called at the start of all the original accept bodies of P. Upon receiving this call the monitor does the following:

1. if the CONSUMER is not in status Calling (e.g. because it issued a conditional or timed entry call) then the actions for entry CALLING are taken. This may cause the SERVER to change status from Accepting to Running.

2. the queue size associated with the entry point is decremented.
3. all of the HERE_FLAGs for the SERVER's entries are cleared, as the server is no longer waiting at any entry.

When receiving the **END_RENDEZVOUS** entry the monitor simply changes the status of CONSUMER back to Running and decrements the NUM_BLOCKED counter. The SERVER and ENTRY_NAME parameters are included for tracing purposes.

The **END_BLOCK** entry has parameters CONSUMER (the task leaving the block) and DEPENDENTS, a list of tasks which are dependent on the scope being left. If some of the DEPENDENTS have not yet terminated, the monitor:

1. for each task on the DEPENDENTS list, sets the PARENT_TASK component of that task's status record to the CONSUMER.
2. sets the CONSUMER's NUM_WAIT_FOR component to the number of tasks on the DEPENDENTS list that have not finished,
3. sets the CONSUMER's status to Block_Waiting
4. increments the NUM_BLOCKED counter.

The **END_TASK** entry is similar to the END_BLOCK entry, except the CONSUMER is placed in status Completed rather than Block_Waiting. If all the dependents have terminated, then the CONSUMER is terminated as well.

3.4.2 GROUP TWO ENTRIES

These entries are used to control diagnostic output from the monitor. Calls to them are placed by the programmer in either the original or transformed Ada source text.

PRINT has no parameters. When the monitor accepts this entry, it prints out its internal picture. Using this, a programmer can get "snapshots" of scheduling states during a computation.

A call to the monitor entry **TRACE** enables (if FLAG is true) or disables (if FLAG is false) trace output for the SUBJECT. When the monitor receives an entry call whose CONSUMER or SERVER parameter is a task with tracing enabled, then the monitor will display the call and its parameters. It is possible to trace all calls to the monitor by using entry TRACE with parameters ALL_TASKS and TRUE. Normal tracing is restored by calling TRACE with ALL_TASKS and FALSE.

3.4.3 GROUP THREE ENTRIES

A deadness error is imminent whenever the DEADLK_FLAG parameter has the value TRUE on completion of a monitor call. Evasive action based on testing this parameter value may be programmed in the original source text (see Chapter 5). The two entries UNBLOCK and QUERY are provided to assist this.

UNBLOCK has a single TASK_ID parameter, SUBJECT. The monitor assumes that the SUBJECT task will *not* proceed with the originally intended action, and updates its picture accordingly, thus "unblocking" the task. UNBLOCK can be severely misused. It should only be called from the task

SUBJECT when the DEADLK_FLAG parameter has been returned true, and SUBJECT is not going to proceed with the tasking statement that has just been indicated by a monitor call.

The entry **QUERY** may be used to help control evasive action routines. A task passes the monitor the SUBJECT, a TASK_ID, and receives information about how that task is blocked. Specifically, the task and entry that the SUBJECT is calling (if any) and the entries that the SUBJECT is accepting (if any) are returned. This entry is intended to allow more intelligent evasive action by giving the task undertaking the evasive action more information about the error.

3.5 ACCURACY OF THE MONITOR PICTURE

The monitor picture may differ from the actual scheduling state of the monitored program. The picture is constructed on the basis of entry calls notifying the monitor of intended task status changes. These entry calls are placed in the monitored program by the preprocessor (Chapter 4). In most cases a notification will be executed before the intended status change (early notification). In some cases, when conditional or timed entry calls are present in the monitored program, the status change may be executed before the monitor is notified (late notification). Also, the underlying scheduling may have the effect that the tasks actually execute status changes in a different order from the notifications. Each of these cases can result in the picture differing from the actual scheduling state.

Example:

T1:

```
...
MONITOR.CALLING(t1, t2, "E");        -- A
t2.E;                                -- B
...
```

T2:

```
...
MONITOR.ACCEPTING(t2, "E");          -- C
accept E;                            -- D
...
```

Before either monitor call, both tasks are in status Running. After t2 has executed statement *C*, it has status Accepting in the picture (but it is actually Running). If t1 now executes statements *A* and *B* then t1 will be Calling in the monitor's picture as well as in the actual scheduling state. This causes t2 to make an indirect status change to Running in the picture. When t2 finally executes *D*, it will remain in status Running since there is a call queued up at the entry. In this example, t2 always had status Running even though it was blocked (Accepting) in the monitor's picture. It is important to show that these temporary inaccuracies do not interfere with the monitor's ability to detect deadness errors.

Here we outline a proof that the monitor correctly detects global blocking situations despite differences between the picture and the actual scheduling state. The proof is based on the following simplifying assumptions, A, about the monitored program:

1. no timed or conditional entry calls are executed.

2. no tasks are aborted.
3. all tasks have the same priority (or all tasks are running on separate CPUs).
4. all tasks in the computation being monitored have already been activated and assigned their IDs.

A *task interaction* is the execution (or partial execution) of a statement that *may* cause a task to change status (directly or indirectly).

Lemma 1: Under A, all tasks notify the monitor immediately before undertaking a task interaction (i.e., if a task notifies the monitor of an impending task interaction, then that interaction will be the next one executed by that task).

Proof: The preprocessor places calls to the monitor immediately before each task interaction except timed/conditional entry calls and selective waits with delay or else parts (see chapter 4). In the absence of timed and conditional entry calls, the execution of a selective wait with a delay or else part is not a task interaction. The task executing the selective wait stays in status Running and any calling task will remain in status Calling. Thus no status changes are caused by a selective wait (under assumptions A) so its execution is not a task interaction.

Lemma 2a: If two task interactions can legally (according to the semantics of the Ada program) be executed in either order, then the same monitor picture results after both notifications are given, regardless of the order.

Lemma 2b: The monitor picture represents the scheduling state of the monitored program if exactly those task interactions that have been signaled to the monitor have taken place.

Lemmas 2a and 2b are implied by the monitor implementation and can be proved by case analysis. However the analysis is tedious and thus is omitted.

Lemma 3: At any point in the execution of a monitored program, there is an actual scheduling state, S, and a monitor picture, P. There is always a legal scheduling under which the execution may continue such that a new scheduling state S' results which is equivalent to the scheduling state that appears in P.

Proof: The actual scheduling state, S, differs from the picture, P, because some tasks have notified the monitor of interactions which they have have not yet executed. By scheduling each of these tasks to run until the notified task interactions have been completed, the resulting scheduling state will agree with the picture. Lemma 1 implies that these tasks can be run and that there will be no additional task interactions. Lemma 2 implies that the resulting picture will agree with the scheduling state. Assumption 3 implies that scheduling (running) only certain tasks is legal.

Lemma 4: Any notification of an impending task interaction is given by a task which has status Running in the monitor picture.

Proof: Tasks can become blocked only by making direct status changes. Any task which is blocked in the monitor picture is either blocked in the actual program or has already notified the monitor of a task interaction that will block it (Lemmas 1 and 2). In either case, it is impossible for the task to issue another notification to the monitor before actually blocking.

Lemma 5: Any global blocking state is present in the picture before it occurs in the actual computation.

Proof (by contradiction): Assume a global blocking situation is present in the computation but not the picture. Then by lemma 3, there is a continuation of the computation which causes the scheduling state to agree with the picture, and thus be unblocked.

Lemma 6: If the monitor picture shows a global blocking situation, B, then the same global blocking situation will occur in a scheduling state of the computation.

Proof: All tasks in the computation (assumption 4) have issued notification of an interaction by which they undergo a direct change to a blocked status. By Lemma 1 every task that is run must execute that interaction before any other interaction (that might unblock a task). Therefore, under any scheduling the global blocking situation will occur.

These arguments can be extended to cover the presence of task activation and timed or conditional entry calls. Similar arguments showing that the picture correctly reflects circular deadlock situations can also be given.

4. PREPROCESSOR

This chapter describes the preprocessor. The purpose of the preprocessor is to introduce communication between the tasks of P and the monitor so that the monitor is informed of any task status change in P. The resulting monitored program is denoted by P'.

The preprocessor applies a sequence of textual transformations. Each transformation introduces new declarations or statements. The transformations can be broken down into atomic steps describable in a formalism similar to the presentation in [2]. However formal description of many details (e.g. transformations for composite data structures containing tasks, and for expressions invoking tasking) is complex. Therefore we have chosen to give an informal description. We describe the preprocessor as a sequence of five passes. First the monitor declaration and body are placed at the beginning of the declarative part of the main program of P. Following this, each succeeding pass is then assumed to take its input from the output of the preceding pass. Each section of this chapter describes a pass (4.1 - first pass, 4.2 - second pass, etc.). We will use P*k* to designate the output from the *k*th pass, thus P2 is the output from the transformations described in Section 4.2.

The transformations set up a *correspondence* (Section 2.3) between P and P' which is also described informally below.

Notes:
Only the original rendezvous attempts between tasks in P are monitored; rendezvous with the monitor itself are not monitored. All identifiers introduced by the preprocessor, e.g. type names and variables, are assumed not to clash with the identifiers in P.

4.1 INTRODUCTION OF TASK ID'S

Passes 1 and 2 introduce task IDs into the monitored program. Pass 1 introduces data structures to store IDs and also new parameters, entry procedures, and accept statements to communicate IDs. Pass 2 introduces code to initialize IDs. The resulting program after passes 1 and 2 has the following properties: *(1)* every active task has a unique ID, *(2)* a calling task can always access the called task's ID, *(3)* a task can access its own ID, *(4)* within every scope the ID of the currently executing task can be accessed, *(5)* whenever an entry is called the ID of the caller is passed to the called task, and *(6)* the monitor associates an identifier in the source text with each task ID which identifies the task object having that ID.

Pass 1 performs the following seven transformations:

1. a new variable, MY_ID of type TASK_ID is declared at the beginning of the main program and initialized to 0.

2. each task type (Ada 83, 9.1) t, is modified to form a new task type renamed t_TASK, followed by a record type with the original name, t.

   a. the simple name t at the beginning and end of the task unit is changed to t_TASK; within the task unit all occurrences of t that designate the task currently executing the body are changed to t_TASK.
   b. a new entry declaration, "entry SET_ID(N : in TASK_ID);" is inserted into the task type specification.

c. two new declarations, "MY_ID : TASK_ID := NULL_TASK; t_TASK_ID renames MY_ID;" are added at the beginning of the task body.
d. the statement "accept SET_ID(N : in TASK_ID) do MY_ID := N; end SET_ID;" is inserted as the first statement in the task body.
e. a new record type,

```
type t is record
    TASK_OBJ : t_TASK;
    ID       : TASK_ID := NULL_TASK;
end record;
```

is declared immediately following the modified task specification.

3. each task declaration, t, in P is replaced by a task type declaration t_TASK, a record type, t_RECORD, and a record of that type with the name, t. The task type t_TASK is obtained from the original task declaration by modifications similar to those stated in step 1; t_RECORD has two components as above.

4. each occurrence of an untransformed task name, t, is replaced by the corresponding record component, t.TASK_OBJ. Thus, for example, entry calls to a task, t.E say, are replaced by entry calls to the task component of the new record, t.TASK_OBJ.E.

5. a new formal parameter called MY_ID of type TASK_ID is added to every subprogram specification.

6. a new formal parameter called CALLER_ID of type TASK_ID is added to every entry specification.

7. all calls to entries and subprograms are modified appropriately as follows: the TASK_ID parameter of every entry and subprogram call is bound to the value of MY_ID. This is either the value of the local MY_ID variable (if the call is in a task) or the value of the formal TASK_ID parameter, MY_ID (if the call is in a subprogram).

As a result of step 2, all task object declarations of a task type in P will become declarations of objects of a record type in P1.

As a result of steps 2 and 3 all task objects occur as components of records which also contain a TASK_ID component. We will call these *task records*. If the original tasks were components of a data structure, the new task records take their place in the structure as a result of using the names of the original task types or tasks as names for the task record types (step 2) or task records (step 3).

When t_TASK is used within a task body to designate the task currently executing the body, the local t_TASK_ID variable will contain the associated ID. Wherever a task was visible in P, now both the task and its ID are visible.

Notes:
The SET_ID entry and the local MY_ID variable are used to "inform" a task of its own ID when it is activated, and to store that ID.
The Ada semantics do not specify the order of task activation. Therefore at steps 2 and 3 "accept SET_ID ..." is inserted as the first statement of every task body; in pass 2 task ID components of all task records are initialized before any task is informed of its ID by a SET_ID entry call. This "holds up" every task until all ID components are initialized, thus avoiding the possibility that tasks in P' might attempt to access task ID components that are uninitialized.

The purpose of steps 4 - 6 is to ensure that the actual value of the CALLER_ID parameter of any entry call is the ID of the task issuing that call. This in turn requires that a subprogram must be able to access the ID of the task that called it so that if it issues an entry call it can pass this ID to the called task. (Note that a subprogram can be visible to, and thus called by, more than one task.) Hence the TASK_ID parameter must be added to both subprograms and entries.

Correspondence: After pass 1, *correspondences* between text of P and new or modified text of P1 is as follows (text in P that is not affected by the transformations corresponds to the same text in P1):

A task object t in P *corresponds* to the task object component of the record with the same name, t, in P1; i.e., t corresponds to t.TASK_OBJ. A task type t in P *corresponds* to a task type in P1 (called t_TASK) obtained by modifying the declaration of t at step 1 above. The old and new subprogram and entry declarations and calls *correspond*. The new variables MY_ID, entries SET_ID, and new **accept** SET_ID statements have no correspondence in P.

EXAMPLES OF PASS 1 TRANSFORMATIONS

1. A task type declaration is transformed into a task type followed by a record type:
Note: TT1 *corresponds to* TT1_TASK.

ORIGINAL TEXT, P:

```
task type TT1 is
    entry E1;
    entry E2(I : in INTEGER; ...);
end TT1;

task body TT1 is
    ...
begin
    ...
end TT1;
```

TRANSFORMED TEXT, P1:

```
task type TT1_TASK is
    entry SET_ID(N      : in TASK_ID);
    entry E1(CALLER_ID : in TASK_ID);
    entry E2(CALLER_ID : in TASK_ID ; I : in INTEGER; ...);
end TT1_TASK;

type TT1 is
    record
        TASK_OBJ : TT1_TASK;
        ID       : TASK_ID := NULL_TASK;
    end record;

task body TT1_TASK is
    MY_ID : TASK_ID := NULL_TASK;
    ...
begin
    accept SET_ID(N : in TASK_ID) do
        MY_ID := N;
```

```
    end;
    ...
end TT1_TASK;
```

2. All task object declarations become task record object declarations:
Note: A_TASK *corresponds to* A_TASK.TASK_OBJ.

ORIGINAL TEXT, P:

```
A_TASK : TT1;
```

TRANSFORMED TEXT, P1:

```
A_TASK : TT1;
```

3. Declarations of single tasks are transformed into a task type and record type declaration, followed by a record declaration:
Note: T1 *corresponds to* T1.TASK_OBJ.

ORIGINAL TEXT, P:

```
task T1 is
    entry E1;
    entry E2(N : in INTEGER; ...);
end T1;

task body T1 is
    ...
end T1;
```

TRANSFORMED TEXT, P1:

```
task type T1_TASK is
    entry SET_ID (N          : in TASK_ID);
    entry E1     (CALLER_ID : in TASK_ID);
    entry E2     (CALLER_ID : in TASK_ID;
                  N          : in INTEGER; ...);
end T1_TASK;

task body T1_TASK is
    MY_ID : TASK_ID := NULL_TASK;
    ...
begin
    accept SET_ID(N : in TASK_ID) do
        MY_ID := N;
    end SET_ID;
    ...
end T1_TASK;

type T1_RECORD is
    record
        TASK_OBJ : T1_TASK;
        ID       : TASK_ID := NULL_TASK;
```

```
        end record;

    T1 : T1_RECORD;
```

4. Pass 1 transformations modify subprogram and entry declarations and calls:

ORIGINAL TEXT, P:

i.
```
        procedure PROC1 is
            ...
        end PROC1;
```

ii.
```
        function F1(I : in INTEGER)
            return SOME_TYPE is
            ...
        end F1;
```

iii.
```
        PROC1;
```

iv.
```
        K:= F1(J);
```

v.
```
        T1.E2(N);
```

TRANSFORMED TEXT, P1:

i.
```
        procedure PROC1(MY_ID : in TASK_ID) is
            ...
        end PROC1;
```

ii.
```
        function F1(MY_ID : in TASK_ID; I : in INTEGER)
            return SOME_TYPE is
            ...
        end F1;
```

iii.
```
        PROC1(MY_ID);
```

iv.
```
        K := F1(MY_ID, J);
```

v.
```
        T1.TASK_OBJ.E2(MY_ID, N);
```

4.2 INITIALIZATION OF TASK ID'S

Pass 2 accepts as input the result of pass 1 and inserts statements to initialize TASK_ID components and variables. When a task record is declared, the declaring scope must call the monitor to obtain a new ID, initialize the ID field of the task record, and inform the task of its ID. If several tasks are declared in the same declarative part then *all* of the ID record components must be initialized before letting any task proceed, otherwise one of the tasks could access an ID component before it has been initialized.

The pass 2 transformations for initializing the IDs of statically declared tasks in each declarative part are:

1. For each task record declaration a call to MONITOR.NEWTASK is inserted; the string parameter of this call is bound to the task record name and the TASK_ID parameter is the task record ID component. If the declaration is in the declarative part of a subprogram or block the call is placed in the first statement position of that subprogram body or block; if the declaration is in the declarative part of a task body, the call is placed immediately following the accept SET_ID statement of the task body.

2. Immediately following all MONITOR.NEWTASK calls inserted at step 1, calls to the SET_ID entry of the task component of each task record are inserted. The TASK_ID parameter of each SET_ID call is bound to the ID component of the same task record.

If tasks are declared as part of a complex structure (built out of arrays, records, and access types) then pass 2 uses iterative techniques to construct the initialization code for objects of that complex type. For example, task IDs occuring as components of arrays are initialized by for loops iterated over the array index type. Details of these techniques are omitted.

Notes:
Calls inserted by step 1 inform the monitor of the identifier in the source text to be associated with each task (for tracing and debugging); TASK_ID values returned by these calls initialize all task record ID components. The monitor can then associate its own ID for a task with a name for the task in the source text. If a task occurs as a value in a data structure, the name of the global data structure is used, so in general many IDs may be associated with a source text name. As a result of calls inserted at step 2, all tasks now "know" their IDs, and have been "held up" until all visible ID components are initialized.
The task ID initialization presented here initializes all IDs immediately after elaboration of a declarative part. This has the drawback that uninitialized IDs can be accessed if task entry calls are executed during elaboration (such accesses may often indicate a blocked elaboration, but not always). This is the reason why our monitoring method is not correct for programs using tasking during elaboration. An alternative scheme is to initialize TASK_ID components and MY_ID variables at the point of declaration via functions which call the monitor entry NEWTASK. These initializations would be placed in the type declarations (of task record types and task type bodies). This would avoid the above drawback. However, a meaningful source text name for the TASK_NAME parameter cannot usually be given in such default positions.

Correspondence: Text to initialize task IDs added by pass 2, steps 1 and 2 does not correspond to any text in P.

EXAMPLES OF PASS 2 TRANSFORMATIONS

P1 DECLARATIVE PART:

```
T1 : SOME_TASK_RECORD_TYPE;

TASK_ARRAY : array (1 .. 5) of SOME_TASK_RECORD_TYPE;

type TWO_TASKS_TYPE is
    record
        FIRST  : SOME_TASK_RECORD_TYPE;
```

```
        SECOND : SOME_TASK_RECORD_TYPE;
        N      : INTEGER;
    end record;
TWO_TASKS : TWO_TASKS_TYPE;
```

P2 IMMEDIATELY FOLLOWING THE BEGIN OF THE DECLARATIVE REGION:

```
-- Text to initialize all task ID components.

    MONITOR.NEWTASK("T1", T1.ID);

    for I in 1 .. 5 loop
        MONITOR.NEWTASK("TASK_ARRAY", TASK_ARRAY(I).ID);
    end loop;

    MONITOR.NEWTASK("TWO_TASKS.FIRST", TWO_TASKS.FIRST.ID);
    MONITOR.NEWTASK("TWO_TASKS.SECOND", TWO_TASKS.SECOND.ID);

-- Text to inform all tasks of their ID's.

    T1.TASK_OBJ.SET_ID(T1.ID);

    for I in (1 .. 5) loop
        TASK_ARRAY(I).TASK_OBJ.SET_ID(TASK_ARRAY(I).ID);
    end loop;

    TWO_TASKS.FIRST.TASK_OBJ.SET_ID(TWO_TASKS.FIRST.ID);
    TWO_TASKS.SECOND.TASK_OBJ.SET_ID(TWO_TASKS.SECOND.ID);
```

The situation in which a new task is created and activated by an allocator requires special handling in pass 2. If P contains an access type designating a type T with task type components, then P1 will contain an access type designating T which now has task record components. Allocation of an object of type T must not be permitted to make an ID component visible before it is initialized. Our approach is to "hide" such allocators in function calls.

Pass 2 contains a third step:

3. Whenever an access type which designates a type containing task components is declared, pass 2 inserts a new function declaration to be associated with the access type. This function will take as parameter a value of the access type and return the same value. It initializes all task IDs in the object designated by its parameter. Wherever an allocator is called in P1 to create a new object containing task components, pass 2 will substitute a call to this new function in P2 with the value of the allocator call as its actual parameter.

Correspondence: The new functions and calls to them have no correspondence in P1. The allocator calls in P1 *correspond* to the allocator call parameters of the new function calls in P2.

Example:

P1:

```
    type TWO_TASKS_TYPE is
```

```
        record
            FIRST  : SOME_TASK_RECORD_TYPE;
            SECOND : SOME_TASK_RECORD_TYPE;
            N      : INTEGER;
        end record;

    type TWO_TASKS_REF is access TWO_TASKS_TYPE;

    TWO_TASKS_PTR : TWO_TASKS_REF;
        .
        .
        .

    TWO_TASKS_PTR := new TWO_TASKS_TYPE;
```

P2:

```
    type TWO_TASKS_TYPE is
        record
            FIRST  : SOME_TASK_RECORD_TYPE;
            SECOND : SOME_TASK_RECORD_TYPE;
            N      : INTEGER;
        end record;

    type TWO_TASKS_REF is access TWO_TASKS_TYPE;

    function NEW_TWO_TASKS(TEMP : in TWO_TASKS_REF) return TWO_TASKS_REF is

    begin                                   -- Initialize TASK_IDs in TEMP.
        MONITOR.NEWTASK(...);
        MONITOR.NEWTASK(...);
        TEMP.FIRST.SET_ID(...);
        TEMP.SECOND.SET_ID(...);
        return TEMP;
    end NEW_TWO_TASKS;

    TWO_TASKS_PTR : TWO_TASKS_REF;
        .
        .
        .
    TWO_TASKS_PTR := NEW_TWO_TASKS (new TWO_TASKS_TYPE);
```

Note on Example:
The call to NEW_TWO_TASKS "hides" the newly allocated value (of type TWO_TASKS_TYPE) until all task IDs in it have been initialized; thus when TWO_TASKS_PTR can be referenced in P2, the IDs in the designated value will have been initlalized.

4.3 MONITORING OF DEPENDENT TASKS

Detection of dead states which include the inability of tasks to terminate usually requires dependency information. For example, a task moves from status Block_Waiting to status Running only when all of its dependents declared in the block have terminated. Consequently, a task may be dead as a result of a deadness among its dependents.

In order to deal with such situations, the preprocessor declares a variable accessing a list of (dependent) task IDs in each block (i.e., each block in P' that corresponds to a block in P now has a list containing all tasks dependent on that block). At run-time this list is passed to the monitor by the executing task when a new dependent task is activated, or when the executing task has reached the end of that block. Thus, in this present monitor design, updating of dependents lists and checking for termination is done by the monitor itself, but the lists of dependents are stored in the monitored program (see Section 3.3).

Pass 3 of the preprocessor has the following steps:

1. Add the declaration, "DEPENDENT_IDS : MONITOR_DATA_PACKAGE.ID_PTR" at the beginning of each declarative part of P2, except for the new subprograms whose declarations were inserted by pass 2.

2. Add the declaration "ALL_DEPENDENTS : MONITOR_DATA_PACKAGE.ID_PTR" at the beginning of the outermost declarative part of each task body and the main program.

3. Insert a call to MONITOR.ADD_DEPENDENT after each call to the monitor entry NEWTASK. The parameters of each call are: PARENT => MY_ID, SON => out parameter of preceding NEWTASK call, BLOCK_DEPENDENT_LIST => DEPENDENTS_IDS, SON_LIST => ALL_DEPENDENTS.

Notes:
Tasks created by an allocator depend on the block where the access type was declared, so their IDs must be added to the DEPENDENT_IDS list corresponding to that block. In P2 these allocator calls are replaced by calls to a new function associated with the access type. This function is declared immediately following the access type by pass 2. It contains the appropriate NEW_TASK calls. Since pass 3 does not insert a declaration of a local DEPENDENT_IDS in these function bodies, the immediately global DEPENDENTS_IDS variable is visible. This will be the DEPENDENTS_IDS variable associated with the declarative part containing the access type declaration. Therefore, the pass 3 monitor calls to ADD_DEPENDENT placed in the function will pass the DEPENDENTS_IDS variable for the block in which the access type is declared to the monitor.
If a select statement, say, in task T1, has a terminate alternative, then the IDs of all tasks directly dependent on T1, or one of its inner blocks, must be passed to the monitor. The variable ALL_DEPENDENTS designates a list of exactly these IDs.

Correspondence: The text added to P3 in pass 3 does not correspond to text in P2.

4.4 STATUS MONITORING

Pass 4 inserts calls to the monitor entries CALLING, ACCEPTING, SELECTING, START_RENDEZVOUS, END_RENDEZVOUS, END_BLOCK, and END_TASK. These calls inform the monitor of direct and indirect status changes, and associated information arising from rendezvous attempts.

The transformation uses strings derived from source text identifiers as names of task entries. These names are used to notify the monitor which entry of a task is being called and are crucial in the monitor's internal representation of rendezvous statuses. These entry name strings must name exactly one entry in any given task: no entry can be represented by two different strings, and no

string can represent two different entries of the same task. A string could represent several entries, as long as they are all in different tasks. An entry family requires a different string for each member of the family. Finally, the transformation introduces arrays for storing and accessing the names associated with entry families; details of these entry family name arrays are omitted.

4.4.1 THE CALLING ENTRY

Calls to this entry are inserted in a task immediately before an unconditional, untimed entry call. When a call to CALLING is executed, the monitor will change the status of the task to Calling. As soon as this monitor call finishes and the next statement is executed, the task's actual status will be Calling (Section 3.4). Timed and conditional entry calls are *not* monitored because they do not result in the task changing status (unless the call has actually been accepted). The CONSUMER parameter is the ID of the task making the call, i.e., the value of MY_ID. For calls of the form, t.TASK_OBJ.E, the SERVER parameter is the ID component of the called task's task record. The ENTRY_NAME parameter is the string, created by the preprocessor, naming the called entry. The DEADLK_FLAG parameter indicates whether evasive action should be taken to avoid a blocked state.

Note:
For calls of the form, t_TASK.E, the SERVER parameter is given the t_TASK_ID value (recall that as a result of pass 1 such calls will always be within the task body of t_TASK see 4.1, pass 1, step 2a).

Correspondence: The CALLING monitor call does not correspond to code in P3.

Examples:

ENTRY CALLS IN P3:

```
T1.TASK_OBJ.E2(MY_ID, PARAMETER);

T1.TASK_OBJ.ENTRY_FAMILY(EXP)(MY_ID);
```

ASSOCIATED MONITOR CALLS INSERTED BY P4:

```
MONITOR.CALLING(MY_ID, T1.ID, "E2"; DEADLK_FLAG);
T1.TASK_OBJ.E2(MY_ID, PARAMETER);

MONITOR.CALLING(MY_ID, T1.ID, ENTRY_FAMILY_STR(EXP), DEADLK_FLAG);
T1.TASK_OBJ.ENTRY_FAMILY(EXP)(MY_ID);
```

4.4.2 THE ACCEPTING ENTRY

Pass 4 inserts a call to ACCEPTING immediately before each "simple" accept statement that is not a select alternative (preprocessing of select alternatives is described in 4.4.3). The parameters are: MY_ID (server name), the preprocessor string naming the entry being accepted, and DEADLK_FLAG.

Correspondence: The ACCEPTING monitor call does not correspond to code in P3.

Example:

P3:

```
accept E1(CALLER_ID) do
    ...
```

P4:

```
MONITOR.ACCEPTING(MY_ID, "E1", DEADLK_FLAG);
accept E1(CALLER_ID) do
    ...
```

4.4.3 THE SELECTING ENTRY

Before executing a selective wait statement a (server) task must inform the monitor of those entries that can be accepted by that select statement. It must therefore evaluate the guards of the select alternatives, including any delay or terminate alternatives. This evaluation must be done only once. The resulting values are used both to determine the information associated with the new Accepting status (or Select_Terminate status) and to execute the select statement. Pass 4 inserts declarations of new variables to hold the values of the guards, and text to evaluate the select guards and construct the status information for the monitor.

Pass 4 executes the following text transformations for each select statement in P3:

1. the select statement is enclosed in the body of a new block statement.

2. boolean variables TEMP1, TEMP2, . . . are declared locally in the new block, one for each select alternative, and initialized to the guard expression of that alternative, or to TRUE if there is no guard.

3. boolean variables TEMP_DELAY and TEMP_TERMINATE are declared locally after the previous variables. TEMP_DELAY is initialized to TRUE if there is an else part, to the disjunction of the TEMP variables corresponding to delay alternatives, or to FALSE if there is no else part or delay alternatives. TEMP_TERMINATE is initialized to the TEMP variable corresponding to the terminate alternative if there is one and to FALSE otherwise.

4. a variable ENTRY_LIST of type ENTRY_PTR is declared locally and initialized to null.

5. ada text to construct the list of entry names corresponding to open accept alternatives is inserted at the beginning of the local block body (i.e., before the select statement). This text is instantiated from a single text template and performs a computation as follows: if TEMP_DELAY is TRUE it does nothing, (including not calling the monitor); otherwise it builds a list of entry name strings corresponding to the open accept alternatives and then calls the monitor entry, SELECTING, with parameters: MY_ID, ENTRY_LIST, TEMP_TERMINATE, ALL_DEPENDENTS, and DEADLK_FLAG.

6. the boolean conditions in the select alternatives are replaced by the corresponding TEMP variables.

Correspondence: The select statement in P4 corresponds to the original select statement in P3.

The new local block, declarations, and text in P4 has no correspondence in P3, except that calls to functions in the new text correspond to the original calls in guards in P3.

Notes:
If TEMP_DELAY is TRUE the server task cannot enter a blocked state and will remain in status Running.
TEMP_TERMINATE is declared even if there is no terminate alternative so that the preprocessor can use a single text template for computing the list of open entries.
TEMP_DELAY and TEMP_TERMINATE cannot both be true due to Ada rules for select statements.
Construction of the list of entries proceeds as follows: ENTRY_LIST is initialized to null; then for each accept alternative with a true guard condition a new ENTRY_REC record containing the string representing the entry is allocated. If the entry is part of an entry family, its index expression is evaluated at this point (to correspond with the order of evaluation in the Ada semantics). This record is inserted into the list designated by ENTRY_LIST.

Examples:

P3:

```
select
        accept E1(CALLER_ID : in TASK_ID) do
            ...
        end E1;
    or
        accept E2(CALLER_ID : in TASK_ID);
                  I         : in INTEGER) do
            ...
        end E2;
end select;

select
    when FLAG1 =>
        accept E1(CALLER_ID : in TASK_ID) do
            ...
        end E1;
or
    when F(X) => delay 10;
end select;
```

P4:

```
declare
    TEMP1 : BOOLEAN := TRUE;
    TEMP2 : BOOLEAN := TRUE;
    TEMP_DELAY     : BOOLEAN := FALSE;
    TEMP_TERMINATE : BOOLEAN := FALSE;
    ENTRY_LIST     : ENTRY_PTR := null;
begin
    if not TEMP_DELAY then
        if TEMP1 then
            ENTRY_LIST := new ENTRY_REC(NAME => "E1",
                                        NEXT => ENTRY_LIST);
        end if;
        if TEMP2 then
            ENTRY_LIST := new ENTRY_REC(NAME => "E2",
```

```
                                                NEXT => ENTRY_LIST);
        end if;
        MONITOR.SELECTING(MY_ID, ENTRY_LIST, TEMP_TERMINATE,
                          ALL_DEPENDENTS, DEADLK_FLAG);
    end if;
    select
        accept E1(CALLER_ID : in TASK_ID) do
        ...
        end E1;
    or
        accept E2(CALLER_ID : in TASK_ID;
                  I         : in INTEGER) do
        ...
        end E2;
    end select;
end;

declare
    TEMP1            : BOOLEAN := FLAG1;
    TEMP2            : BOOLEAN := F(X);
    TEMP_DELAY       : BOOLEAN := TEMP2;
    TEMP_TERMINATE   : BOOLEAN := FALSE;
    ENTRY_LIST       : ENTRY_PTR := null;
begin
    if not TEMP_DELAY then
        if TEMP1 then
            ENTRY_LIST := new ENTRY_REC(NAME => "E1",
                                        NEXT => ENTRY_LIST);
        end if;

        MONITOR.SELECTING(MY_ID, ENTRY_LIST, TEMP_TERMINATE,
                                             ALL_DEPENDENTS, DEADLK_FLAG);
    end if;

    select
        when TEMP1 =>
            accept E1(CALLER_ID) do
            ...
            end E1;
    or
        when TEMP2 => delay 10;
    end select;
end;
```

Note:
Often text inserted by pass 4 can be omitted. In the first example above, none of the TEMP variables for accept alternatives, nor the corresponding conditional tests on them, are needed. The preprocessor does in fact make some optimizations on the use of the TEMP variables.

4.4.4 THE START_RENDEZVOUS ENTRY

Pass 4 inserts a call to this monitor entry at the beginning of every accept body, including those inside of select statements. The parameters of the call are: CONSUMER => CALLER_ID (a parameter of the entry call), SERVER => MY_ID, ENTRY_NAME => the string associated with the entry being accepted.

Correspondence: This entry call has no corresponding code in P.

4.4.5 THE END_RENDEZVOUS ENTRY

A call to this entry is placed at the end of every accept body (including those in selective wait statements). The parameters of this call are: CALLER_ID, MY_ID, and the string associated with the entry accepted.

Correspondence: This entry call does not correspond to any code in P3.

Examples:

P3:

```
accept E1(CALLER_ID : in TASK_ID);

accept E2(CALLER_ID : in TASK_ID; I : in INTEGER; ...) do
    ...
end E2;
```

P4:

```
accept E1(CALLER_ID : in TASK_ID) do
    MONITOR.START_RENDEZVOUS(MY_ID, CALLER_ID, "E1");
    MONITOR.END_RENDEZVOUS(MY_ID, CALLER_ID, "E1");
end E1;

accept E2(CALLER_ID : in TASK_ID ; I : in INTEGER; ...) do
    MONITOR.START_RENDEZVOUS(MY_ID, CALLER_ID, "E2");
        ...
    MONITOR.END_RENDEZVOUS(MY_ID, CALLER_ID, "E2");
end E2;
```

4.4.6 THE END_TASK ENTRY

A call to this entry is inserted at the end of every task body. The parameters are MY_ID (the ID of the task that is completing), ALL_DEPENDENTS (the IDs of all tasks dependent on the completing task), and DEADLK_FLAG. The value returned for DEADLK_FLAG will indicate whether or not the task will cause a blocked state by completing.

Correspondence: This entry call does not correspond to any code in P.

4.4.7 THE END_BLOCK ENTRY

Calls to this entry are inserted in each inner block and subprogram which has a declarative part. The calls are inserted at: the end of the sequence of statements of the block or subprogram; immediately before each return statement; and immediately before each goto statement transferring control out of the block. In addition, to allow for exceptions raised in the handler itself, each exception handler is modified as follows:

P3:

```
when EXCEPTION_NAME =>
    ...                                -- Sequence of statements.
```

P4:

```
when EXCEPTION_NAME =>
    begin
        ...                            -- Sequence of statements.
        MONITOR.END_BLOCK(...);
    exception
        when others =>
            MONITOR.END_BLOCK(...);
            raise;
    end;
```

Finally, if there is no handler with an others exception choice, then the following exception handler is placed in the block (or subprogram);

```
exception
    when others => MONITOR.END_BLOCK(...); raise;
```

The parameters of this call are the same as for END_TASK, except that the local DEPENDENT_IDS variable takes the place of ALL_DEPENDENTS.

Correspondence: The added text does not correspond to any code in P3.

Notes:
These transformations ensure that the END_BLOCK entry is called before the block is left, even if exceptions are raised or propagated in exception handlers. Not all of these transformations are done by the current pre-processor.

4.5 FUNCTION CALLS IN TASKING STATEMENTS

The transformations in pass 4 are inadequate when parameters of tasking statements contain function calls since evaluation of these parameters might also involve tasking.

Example of inadequacy:

P:

```
...
function F1(ARG : in INTEGER) return INTEGER is
    T3 : SOME_TASK_RECORD_TYPE;
begin
    ...                                    -- Body of F1.
end F1;
    ...
T1.E2(F1(X));
```

P4:

```
...
function F1(MY_ID : in TASK_ID; ARG : in INTEGER) return INTEGER is
    T3 : SOME_TASK_RECORD_TYPE;
    DEPENDENT_IDS : ID_PTR;

begin
    MONITOR.NEW.TASK(...);
    MONITOR.ADD_DEPENDENT(...);
    T3.TASK._OBJ.SET_ID(...);
    ...                                    -- Previous body of F1.
    MONITOR.END_BLOCK(MY_ID, DEPENDENT_IDS, DEADLK_FLAG);
end F1;
...

MONITOR.CALLING(MY_ID, T1.ID, "E2");
                                           -- A
T1.TASK_OBJ.E2(MY_ID, F1(MY_ID, X));
```

At point *A* in the above example, the executing task is in status Calling in the monitor's picture (calling T1). However, when the call to F1 is executed, this executing task could also be put into status Block_Waiting waiting for tasks dependent on F1 to terminate. Currently, this will confuse the monitor and may lead it to falsely detect a global blocking situation, or not detect an actual one. The preprocessor therefore moves all function calls out of tasking statements. This requires additional temporary variables to hold the values of parameter expressions and intermediate values.

Example:

P4:

```
MONITOR.CALLING(MY_ID, T1.ID, "E2");
T1.TASK_OBJ.E2(MY_ID, F1(MY_ID, X));
```

P5:

```
TEMP1 := F1(MY_ID, X);
MONITOR.CALLING(MY_ID, T1.ID, "E2");
T1.TASK_OBJ.E2(MY_ID, TEMP1);
```

Correspondence: The added assignment statements do not correspond to code in P4. However, code of the function bodies executed during the evaluation of the right sides of these statements will correspond to code in P4.

5. EVASIVE ACTION

A task may be programmed to take evasive action to avoid a deadness error detected by a monitor. In this chapter, we outline three techniques for programming evasive action in Ada. These techniques rely on Ada exception propagation and implementation defined pragmas. The implementation defined pragmas are used solely to facilitate the preprocessing of evasive action text. Exception propagation is assumed to be the method used by a monitor to trigger evasive action (see note below). Exceptions signifying the detection of different kinds of dead states are assumed to be declared in the visible part of the MONITOR_DATA_PACKAGE.

Example:

```
GLOBAL_BLOCKING, CIRCULAR_DEADLOCK,
DEPENDENTS_BLOCKED, LOCAL_BLOCKING : exception;
```

The monitor will propagate an exception to the task whose intended direct status change completes a dead state. The task receiving the exception may be no more responsible for the dead state than any other task in the system, however its communication to the monitor caused a dead state to appear in the monitor's picture. In more sophisticated monitoring systems, exceptions may be propagated to other tasks as well as the "final" one.

The three evasive action paradigms are simply templates. The programmer must predict what dead states are possible and include exceptions handlers to take appropriate action. We expect that different applications will require radically different evasion strategies.

Note:
Due to deficiencies in the Adam compiler, our present monitor uses parameters, DEADLK_FLAG, instead of exceptions to indicate dead states. However it is a straightforward exercise to replace the present parameter-based implementation by the (preferred) exception propagation techniques outlined here. Somewhat more sophisticated query facilities than are implemented in our present monitor may be needed. An un-preprocessed evasive action program must use the MONITOR_DATA_PACKAGE if it is to be a legal Ada83 program so that the new exceptions are visible.

5.1 PASSIVE EVASION

Passive evasion is appropriate when a dead state arises because of a scarcity of resources. A task undertaking passive evasion "backs up" its computation and releases recently acquired resources. This allows other tasks to obtain the resources and (hopefully) complete their computations.

Programming passive evasion simply requires placing a suspect tasking statement in a local block with an exception handler containing the evasive action; resources are released and control is returned to an appropriate previous point. The pragma, PASSIVE_EVASION results in insertion of the correct monitor call during preprocessing.

Example

ORIGINAL PROGRAM P BEFORE PREPROCESSING:

```
...                                     -- Save computation state here in order to retry if
<<CHECK_POINT>>                         -- evasive action is taken.
...
RESOURCE1.REQUEST;
...
begin                                   -- Local block surrounding suspect statement.
    RESOURCE2.REQUEST;                  -- This call is suspect and may
exception                               -- lead to GLOBAL_BLOCKING
    when GLOBAL_BLOCKING =>
        pragma PASSIVE_EVASION;         -- The pragma is recognized by
                                        -- the preprocessor; a monitor call to
                                        -- UNBLOCK will be inserted.
        ...                             -- Restore saved computation prior
                                        -- to evasive action.
        RESOURCE1.RELEASE;              -- evasive action: release resource 1.
        goto CHECK_POINT;               -- Try again.
end;
...
```

RESULTING PREPROCESSED PROGRAM P':

```
...                                     -- Save computation.
<<CHECK_POINT>>
...
MONITOR.CALLING(MY_ID, RESOURCE1.ID, "REQUEST");
RESOURCE1.TASK_OBJ.REQUEST(MY_ID);
...
begin
    MONITOR.CALLING(MY_ID, RESOURCE2.ID, "REQUEST");
    RESOURCE2.TASK_OBJ.REQUEST(MY_ID);
exception
    when GLOBAL_BLOCKING =>
        MONITOR.UNBLOCK(MY_ID);         -- "Take back" intended call to RESOURCE2.
        ...                             -- Restore saved computation.
        MONITOR.CALLING(MY_ID, RESORCE1.ID, "RELEASE");
        RESORCE1.TASK_OBJ.RELEASE(MY_ID);
        goto CHECK_POINT;
end;
...
```

5.2 ACTIVE EVASION

A task taking active evasion continues with its execution after interacting with another task. This interaction is intended to free up the other task, thus avoiding the dead state.

The task receiving the exception will issue an entry call, accept an entry, or abort another task based on information received from the monitor (Section 3.4.3). If the dead state is a circular deadlock, then accepting an entry call may be appropriate action. If the dead state involves the inability to leave an inner block, then aborting a dependent task may be necessary.

Example of Evasion of Circular Deadlock:

ORIGINAL PROGRAM P BEFORE PREPROCESSING:

```
...
<<L>>
begin
    T1.E(...)                          -- This call may complete a dead state.
exception
    when CIRCULAR_DEADLOCK =>
        pragma ACTIVE_EVASION;         -- This pragma inserts a call to UNBLOCK.
        pragma CIRCULAR_DEADLOCK_QUERY(MY_ID, CALLED_ENTRY);
        -- This pragma inserts a call to a monitor query entry for circular deadlock; as a
        -- result, CALLED_ENTRY is bound to the entry of MY_ID called in the circular
        -- deadlock.
        select
            when CALLED_ENTRY = "E1" =>
                accept E1(...) do
                    ...
                end E1;
        or
            when CALLED_ENTRY = "E2" =>
                accept E2(...) do
                    ...
                end E2;
        or
            ...
        end select;
    end;
    goto <<L>>;                        -- Resume attempted computation.
end;
```

Note:
The monitor query facility used in the example could easily be provided in our current monitor implementation. The resumed computation may have to repeat the evasion since the task in the circle of calling tasks may not be the first task in the entry queue.

5.3 CATASTROPHE

In a catastrophe there is no hope of "the offending task(s)" continuing to function usefully. If this kind of error is signalled the offender will simply report diagnostics and possibly transmit warnings to other tasks in the program. The reporting can be directed by "querying" the monitor.

Example:

```
task body T is
    ...
begin
    ...
    MONITOR.CALLING(MY_ID, S_ID, "E");
    S.E;
    ...
```

```
exception
    when GLOBAL_BLOCKING => ...
                                        -- Report local conditions and then die
                                        -- gracefully; do not continue.
                                        -- The monitor will automatically give a
                                        -- description of the globally blocked state.
end T;
```

6. EXAMPLES

In this chapter we give examples of the preprocessing transformation, of the monitor's diagnostic output describing deadness errors, and of evasive action.

6.1 A DINING PHILOSOPHERS PROGRAM

The following example is the version of the dining philosophers problem with a potential blocking error given by Hoare in his paper on Communicating Sequential Processes. The example gives the original Ada text, the preprocessed text, and diagnostics from the monitor describing the blocking error when it occurred.

Blocking can occur as follows.

> All five philosophers can enter the room, sit down at the table and pickup one fork. All forks will then be in Accepting status waiting for a PUTDOWN, while all philosophers will be in Calling status having called PICKUP for their second fork. The table will be waiting for either of its entries to be called.

Whether or not this situation will happen depends on the underlying scheduling. The error may never occur or may occur almost immediately, depending on the run-time task supervisor. This is illustrated by the delay statements in the Philosopher task body. If the delay before picking up the second fork is removed, the blocked state will never occur when the program is run with the task supervisor package in the standard Adam environment [4]; with this delay, the tasks block *before* any philosopher eats.

```
with DTTY_IO;
use  DTTY_IO;

procedure ROOM is                       -- The cast of actors: FORKS,
pragma MAIN;                            -- PHILOSOPHERS, and TABLE.

    task type FORK is
        entry PICKUP;
        entry PUTDOWN;
    end FORK;

    task TABLE is
        entry SITDOWN(I : out INTEGER);
        entry GETUP(I : in INTEGER);
    end TABLE;

    task type PHILOSOPHER;

    type SET_OF_FORKS is array (0 .. 4) of FORK;
    FORKS : SET_OF_FORKS;
                                        -- The scripts: the bodies of the actors.

    task body FORK is
    begin
        loop
            accept PICKUP;
```

```
            accept PUTDOWN;
        end loop;
    end FORK;

    task body TABLE is

        type SEAT_ARRAY is array (0 .. 4) of BOOLEAN;
        SEATS : SEAT_ARRAY := (others => TRUE);
                                        -- True means unoccupied.
    begin
        loop
            select
                accept SITDOWN(I : out INTEGER) do
                    for J in 0..4 loop
                        I := J;
                        exit when SEATS(J);
                    end loop;
                    SEATS(I) := FALSE;
                end;
            or
                accept GETUP(I : in INTEGER) do
                    SEATS(I) := TRUE;
                end;
            end select;
        end loop;

    end TABLE;

    task body PHILOSOPHER is
        SEAT : INTEGER;
    begin
        loop
            delay 1;                    -- Delays are for thought. If a large enough
            TABLE.SITDOWN(SEAT);        -- delay is placed between picking up the
            FORKS(SEAT).PICKUP;         -- two forks then the blocked state occurs;
            delay 2;                    -- If not, the philosophers don't block.
            FORKS((SEAT+1) mod 5).PICKUP;
                                        -- This illustrates the dependence of
                                        -- the error on the run-time supervision.
            delay 1;
            FORKS(SEAT).PUTDOWN;
            FORKS((SEAT+1) mod 5).PUTDOWN;
            TABLE.GETUP(SEAT);
        end loop;
    end PHILOSOPHER;

    SOCRATES, PLATO, ARISTOTLE, MARX, RUSSELL : PHILOSOPHER;

begin                                   -- The five forks, five philosophers, and the
    null;                               -- table are all activated at this point.
end ROOM;
```

6.1.1 THE PREPROCESSED AND MONITORED DINING PHILOSOPHERS

The source code of the Dining Philosophers program after preprocessing is given. The reader should compare this version with the original text in Section 6.1 and with the descriptions of preprocessing in Chapter 4. The number of statements inserted by preprocessing is a function of the number of original tasking statements. Here, since the dining philosophers example consists mainly of tasking statements, the output program is significantly longer.

```
with MONITOR_DATA_PACKAGE; use MONITOR_DATA_PACKAGE;
with DTTY_IO;
use  DTTY_IO;
-- The cast of actors: FORKS, PHILOSOPHERS, and TABLE.
procedure  ROOM  is

task MONITOR is
    ...
end;
task body MONITOR is
    ...                                    -- The MONITOR task described in Section 3.
end MONITOR;

    pragma MAIN;
    -- Variables and new type declarations are inserted by pass 1, (Section 4.1)
    -- to introduce task ids; compare with declarations in Section 6.1.
    MY_ID           : constant TASK_ID := 0;
    DEPENDENT_IDS   : ID_PTR;
    ALL_DEPENDENTS  : ID_PTR;
    DEADLOCK_FLAG   : BOOLEAN;

    task type FORK_TASK is
        entry SET_ID    (N         : in TASK_ID);
        entry PICKUP    (CALLER_ID : in TASK_ID);
        entry PUTDOWN   (CALLER_ID : in TASK_ID);
    end FORK_TASK;

    type FORK is
        record
            TASK_OBJ : FORK_TASK;
            ID       : TASK_ID := NULL_TASK;
        end record;

    task type TABLE_TASK is
        entry SET_ID    (N         : in TASK_ID);
        entry SITDOWN   (CALLER_ID : in TASK_ID; I : out INTEGER);
        entry GETUP     (CALLER_ID : in TASK_ID; I : in INTEGER);
    end TABLE_TASK;

    type TABLE_RECORD is
        record
            TASK_OBJ : TABLE_TASK;
            ID       : TASK_ID := NULL_TASK;
        end record;

    TABLE : TABLE_RECORD;
```

```
task type PHILOSOPHER_TASK is
    entry SET_ID(N : in TASK_ID);
end;

type PHILOSOPHER is
    record
        TASK_OBJ : PHILOSOPHER_TASK;
        ID       : TASK_ID := NULL_TASK;
    end record;

type SET_OF_FORKS is array (0 .. 4) of FORK;
FORKS : SET_OF_FORKS;

-- The scripts: the bodies of the actors.

task body FORK_TASK is
    MY_ID             : TASK_ID := NULL_TASK;
    DEPENDENT_ID      : ID_PTR;
    ALL_DEPENDENTS    : ID_PTR;
    DEADLOCK_FLAG     : BOOLEAN;
begin
    accept SET_ID(N   : in TASK_ID) do
        MY_ID := N;              -- Task waits until its ID is initialized
    end;                         -- (Section 4.1)
loop

    MONITOR.ACCEPTING(MY_ID, "PICKU", DEADLOCK_FLAG);
    accept PICKUP(CALLER_ID : in TASK_ID) do
        MONITOR.START_RENDEZVOUS(CALLER_ID, MY_ID, "PICKU");
        MONITOR.END_RENDEZVOUS(CALLER_ID, MY_ID, "PICKU");
    end;

    MONITOR.ACCEPTING(MY_ID, "PUTDO", DEADLOCK_FLAG);
    accept PUTDOWN(CALLER_ID : in TASK_ID) do
        MONITOR.START_RENDEZVOUS(CALLER_ID, MY_ID, "PUTDO");
        MONITOR.END_RENDEZVOUS(CALLER_ID, MY_ID, "PUTDO");
    end;

end loop;

    MONITOR.END_TASK(MY_ID, ALL_DEPENDENTS, DEADLOCK_FLAG);
end FORK_TASK;

task body TABLE_TASK is
    MY_ID             : TASK_ID := NULL_TASK;
    DEPENDENT_IDS     : ID_PTR;
    ALL_DEPENDENTS    : ID_PTR;
    DEADLOCK_FLAG     : BOOLEAN;

    type SEAT_ARRAY is array (0 .. 4) of BOOLEAN;
    SEATS : SEAT_ARRAY := (others => TRUE);
begin
    accept SET_ID(N : in TASK_ID) do
        MY_ID := N;
    end;
```

```
        loop
                declare
                    TEMP_1          : BOOLEAN   := TRUE;
                    TEMP_2          : BOOLEAN   := TRUE;
                    TEMP_DELAY      : BOOLEAN   := FALSE;
                    TEMP_TERMINATE  : BOOLEAN   := FALSE;
                    ENTRY_LIST      : ENTRY_PTR := null;
                end

                if not TEMP_DELAY then
                    if TEMP_1 then
                        ENTRY_LIST := new ENTRY_REC(NAME => "SITDO",
                                                    NEXT => ENTRY_LIST);
                    end if;
                    if TRUE then
                        ENTRY_LIST := new ENTRY_REC(NAME =>"GETUP",
                                                    NEXT => ENTRY_LIST);
                    end if;
                    MONITOR.SELECTING(MY_ID, ENTRY_LIST, TEMP_TERMINATE;
                                      ALL_DEPENDENTS, DEADLOCK_FLAG);
                end if;
                select
                    when TEMP_1 =>
                    accept SITDOWN(CALLER_ID : in TASK_ID;
                                   I         : out INTEGER) do
                        MONITOR.START_RENDEZVOUS(CALLER_ID, MY_ID,"SITDO");
                        for J in 0 .. 4 loop
                            I := J;

                            exit when SEATS(J);

                        end loop;
                        SEATS(I) := FALSE;
                        MONITOR.END_RENDEZVOUS(CALLER_ID, MY_ID, "SITDO");
                    end;

                or
                    when TEMP_2 =>
                    accept GETUP(CALLER_ID : in TASK_ID;
                                 I         : in INTEGER) do
                        MONITOR.START_RENDEZVOUS(CALLER_ID, MY_ID,"GETUP");
                        SEATS(I) := TRUE;
                        MONITOR.END_RENDEZVOUS(CALLER_ID, MY_ID,"SITDO");
                    end;

                end select;

            end loop;

            MONITOR.END_TASK(MY_ID, ALL_DEPENDENTS, DEADLOCK_FLAG);
        end TABLE_TASK;

    task body PHILOSOPHER_TASK is
        MY_ID           : TASK_ID := NULL_TASK;
        DEPENDENT_IDS   : ID_PTR;
```

```
        ALL_DEPENDENTS  : ID_PTR;
        DEADLOCK_FLAG   : BOOLEAN;
        SEAT            : INTEGER;
    begin
        accept SET_ID(N : in TASK_ID) do
            MY_ID := N;
        end;
        loop
            delay 1;

            MONITOR.CALLING(MY_ID, TABLE.ID, "SITDO", DEADLOCK_FLAG);
            TABLE.TASK_OBJ.SITDOWN(MY_ID,SEAT);

            MONITOR.CALLING(MY_ID, FORKS(SEAT).ID, "PICKU",DEADLOCK_FLAG);
            FORKS(SEAT).TASK_OBJ.PICKUP(MY_ID);
            delay 2;

            MONITOR.CALLING(MY_ID, FORKS((SEAT+1) mod 5).ID, "PICKU",
                                                            DEADLOCK_FLAG);
            FORKS((SEAT+1) mod 5).TASK_OBJ.PICKUP(MY_ID);
            delay 1;

            MONITOR.CALLING(MY_ID, FORKS(SEAT).ID, "PUTDO", DEADLOCK_FLAG);
            FORKS(SEAT).TASK_OBJ.PUTDOWN(MY_ID);

            MONITOR.CALLING(MY_ID, FORKS((SEAT+1) mod 5).ID, "PUTDO",
                                                            DEADLOCK_FLAG);
            FORKS((SEAT+1) mod 5).TASK_OBJ.PUTDOWN(MY_ID);

            MONITOR.CALLING(MY_ID, TABLE.ID, "GETUP", DEADLOCK_FLAG);
            TABLE.TASK_OBJ.GETUP(MY_ID, SEAT);

        end loop;

        MONITOR.END_TASK(MY_ID, ALL_DEPENDENTS, DEADLOCK_FLAG);
    end PHILOSOPHER_TASK;

    SOCRATES  : PHILOSOPHER;
    PLATO     : PHILOSOPHER;
    ARISTOTLE : PHILOSOPHER;
    MARX      : PHILOSOPHER;
    RUSSELL   : PHILOSOPHER;
begin

-- Monitor calls inserted by pass 2 (Section 4.2) to initialize all task ids in task records,
-- and track task dependencies (Section 4.3)
    MONITOR.NEWTASK("TABLE", TABLE.ID);
    MONITOR.ADD_DEPENDENT(MY_ID, TABLE.ID, DEPENDENT_IDS, ALL_DEPENDENTS);
    for MON_I1 in 0 .. 4 loop
        MONITOR.NEWTASK("FORKS", FORKS(MON_I1).ID);
        MONITOR.ADD_DEPENDENT(MY_ID, DEPENDENT_IDS, FORKS(MON_I1).ID);
    end loop;
    MONITOR.NEWTASK("SOCRA", SOCRATES.ID);
    MONITOR.ADD_DEPENDENT(MY_ID,SOCRATES.ID,DEPENDENT_IDS,ALL_DEPENDENTS);
    MONITOR.NEWTASK("PLATO", PLATO.ID);
```

```
    MONITOR.ADD_DEPENDENT(MY_ID, PLATO.ID, DEPENDENT_IDS, ALL_DEPENDENTS);
    MONITOR.NEWTASK("ARIST", ARISTOTLE.ID);
    MONITOR.ADD_DEPENDENT(MY_ID,ARISTOTLE.ID,DEPENDENT_IDS,ALL_DEPENDENTS);
    MONITOR.NEWTASK("MARX ", MARX.ID);
    MONITOR.ADD_DEPENDENT(MY_ID, MARX.ID, DEPENDENT_IDS, ALL_DEPENDENTS);
    MONITOR.NEWTASK("RUSSE", RUSSELL.ID);
    MONITOR.ADD_DEPENDENT(MY_ID, RUSSELL.ID, DEPENDENT_IDS, ALL_DEPENDENTS);

    -- SET_ID calls to inform each task of its ID inserted by pass 2 (Section 4.2).

    TABLE.TASK_OBJ.SET_ID(TABLE.ID);
    for MON_I1 in 0 .. 4 loop
        FORKS(MON_I1).TASK_OBJ.SET_ID(FORKS(MON_I1).ID);
    end loop;
    SOCRATES.TASK_OBJ.SET_ID(SOCRATES.ID);
    PLATO.TASK_OBJ.SET_ID(PLATO.ID);
    ARISTOTLE.TASK_OBJ.SET_ID(ARISTOTLE.ID);
    MARX.TASK_OBJ.SET_ID(MARX.ID);
    RUSSELL.TASK_OBJ.SET_ID(RUSSELL.ID);

    null;

    MONITOR.END_TASK(MY_ID, ALL_DEPENDENTS, DEADLOCK_FLAG);
end ROOM;
```

6.1.2 DIAGNOSTIC DESCRIPTION OF THE DINING PHILOSOPHER'S DEAD STATE

Below is the description of a global blocking state given by the monitor.

Key: In descriptions of Accepting status, each entry name is followed by it's queue size (an integer) and a "*" if the task is in a status accepting that entry.

```
    **MON**  GLOBAL DEADNESS HAS BEEN DETECTED
    **MON**    TASK INFORMATION

    0 MAIN  is Block_Waiting on 11 tasks.
        Its entries are:
                <NONE>
      Its father is: -1

-- This description indicates that the table task is in Accepting status,
-- accepting either entry, and neither entry has been called.

    1 TABLE is accepting
        Its entries are:
                SITDO(0*)  GETUP(0*)
      Its father is: 0

-- Fork indicated as task 2 is in status accepting PUTDOWN which has no callers,
-- while some task has called PICKUP.

    2 FORKS is accepting
        Its entries are:
```

```
            PUTDO(0*)  PICKU(1)
  Its father is: 0

3 FORKS is accepting
    Its entries are:
            PUTDO(0*)  PICKU(1)
  Its father is: 0

4 FORKS is accepting
    Its entries are:
            PUTDO(0*)  PICKU(1)
  Its father is: 0

5 FORKS is accepting
    Its entries are:
            PUTDO(0*)  PICKU(1)
  Its father is: 0

6 FORKS is accepting
    Its entries are:
            PUTDO(0*)  PICKU(1)
  Its father is: 0
```

-- SOCRATES is task 7; it has called task 3 (a fork) entry PICKUP;
-- we can see above that task 3 is accepting PUTDOWN.

```
7 SOCRA is calling task number 3 at entry PICKU
    Its entries are:
            <NONE>
  Its father is: 0

8 PLATO is calling task number 4 at entry PICKU
    Its entries are:
            <NONE>
  Its father is: 0

9 ARIST is calling task number 5 at entry PICKU
    Its entries are:
            <NONE>
  Its father is: 0

10 MARX is calling task number 6 at entry PICKU
    Its entries are:
            <NONE>
  Its father is: 0

11 RUSSE is calling task number 2 at entry PICKU
    Its entries are:
            <NONE>
  Its father is: 0
**MON**  end of dead state description.
```

6.2 THE EVASIVE ACTION PHILOSOPHER TASK

The following is an example of a philosopher task with additional evasive action capability. If the task receives a warning from the monitor after informing it that the next action is to pickup its right hand fork, the evasive action will be to putdown the lefthand fork. It will then attempt to eat again as before.

This can be programmed using Paradigm 1, Section 5. It is assumed that this source text will be preprocessed and monitor calls placed as usual, including the evasive action text.

```
    task body PHILOSOPHER is
        SEAT : INTEGER;
    begin
        loop
            delay 1;
            TABLE.SITDOWN(SEAT);
            FORKS(SEAT).PICKUP;
            delay 1;
            begin
-- This call might propagate GLOBAL_BLOCKING, — see Note below.
                FORKS((SEAT+1) mod 5).PICKUP;
            exception
                when GLOBAL_BLOCKING =>
                    FORKS(SEAT).PUTDOWN; -- Evasive action: put down left
                                         -- hand fork.
                    FORKS(SEAT).PICKUP;  -- Try to pick up both forks again.
                    FORKS((SEAT+1) mod 5).PICKUP;
                                         -- May get same error again here.
             end;
             delay 1;
             FORKS(SEAT).PUTDOWN;
             FORKS((SEAT+1) mod 5).PUTDOWN;
             TABLE.GETUP(SEAT);
        end loop;
    exception
-- The evasive action did not solve the problem, so give diagnostics and terminate.
        when GLOBAL_BLOCKING =>
            ...
    end PHILOSOPHER;
```

Note on Example:
When this text is preprocessed, all tasking statements including those in the exception handler will be monitored. Thus the exceptional behavior may also result in a monitor error message. In this case, the outermost exception handler will be executed. This handler would probably contain calls to the monitor to print diagnostics.

Since the Adam compiler does not implement exception propagation during task rendezvous (e.g. rendezvous with the monitor task), evasive action in our experiments is programmed using monitor calls and the value of a parameter, MON_DEAD_FLAG. The evasive action has to be inserted after the program has been preprocessed since we do not want the evasive action monitor calls to be monitored.

6.2.1 TEXT OF AN EVASIVE ACTION PHILOSOPHER AFTER PREPROCESSING.

```
task body PHILOSOPHER_TASK is
    MY_ID          : TASK_ID := NULL_TASK;
    DEPENDENT_IDS  : MONITOR_DATA_PACKAGE.ID_PTR;
    ALL_DEPENDENTS : MONITOR_DATA_PACKAGE.ID_PTR;
    SEAT           : INTEGER;
begin
    accept SET_ID(N : in TASK_ID) do
        MY_ID := N;
    end;
    loop
        delay 1;

        MONITOR.CALLING(MY_ID, TABLE.ID, "SITDO", DEADLOCK_FLAG);
        TABLE.TASK_OBJ.SITDOWN(MY_ID,SEAT);

        MONITOR.CALLING(MY_ID, FORKS(SEAT).ID,"PICKU", DEADLOCK_FLAG);
        FORKS(SEAT).TASK_OBJ.PICKUP(MY_ID);
        delay 1;
        -- This call is suspect and may require evasive action.

        MONITOR.CALLING(MY_ID, FORKS((SEAT+1) mod 5).ID, "PICKU",
                                                        DEADLOCK_FLAG);
        -- Passive evasive action: release resources and then retry.
        if DEADLOCK_FLAG then
            MONITOR.TRACE(ALL_TASKS, TRUE);
            -- Inform monitor of intended evasion.
            MONITOR.UNBLOCK(MY_ID);

            MONITOR.CALLING(MY_ID,FORKS(SEAT).ID,"PUTDO",DEADLOCK_FLAG);
            FORKS(SEAT).TASK_OBJ.PUTDOWN(MY_ID);

            MONITOR.CALLING(MY_ID,FORKS(SEAT).ID,"PICKU",DEADLOCK_FLAG);
            FORKS(SEAT).TASK_OBJ.PICKUP(MY_ID);

            MONITOR.CALLING(MY_ID, FORKS((SEAT+1) mod 5).ID,"PICKU",
                                                        DEADLOCK_FLAG);

        end if;

        FORKS((SEAT+1) mod 5).TASK_OBJ.PICKUP(MY_ID);
        delay 1;

        MONITOR.CALLING(MY_ID, FORKS(SEAT).ID,"PUTDO", DEADLOCK_FLAG);
        FORKS(SEAT).TASK_OBJ.PUTDOWN(MY_ID);

        MONITOR.CALLING(MY_ID, FORKS((SEAT+1) mod 5).ID, "PUTDO",
                                                        DEADLOCK_FLAG);
        FORKS((SEAT+1) mod 5).TASK_OBJ.PUTDOWN(MY_ID);

        MONITOR.CALLING(MY_ID, TABLE.ID, "GETUP", DEADLOCK_FLAG);
        TABLE.TASK_OBJ.GETUP(MY_ID,SEAT);

    end loop;
```

```
        MONITOR.END_TASK(MY_ID, MON_DEPEND_ID, DEADLOCK_FLAG);
    end PHILOSOPHER_TASK;
```

6.2.2 ACTION OF DINING PHILOSOPHERS WITH EVASIVE ACTION

Below is a trace of activity by the evasive version of the dining philosophers. First the monitor description of an imminent dead state is given. A philosopher task is warned, and a trace of its evasive action and subsequent "normal" activity then follows.

Key: See Example 6.1.

```
**MON**  GLOBAL DEADLOCK HAS BEEN DETECTED
**MON**    TASK INFORMATION
0 MAIN  is Block_Waiting on 11 tasks.
    Its entries are:
          <NONE>
  Its father is: -1

1 TABLE is accepting
    Its entries are:
          SITDO(0*)  GETUP(0*)
  Its father is: 0

2 FORKS is accepting
    Its entries are:
          PUTDO(0*)  PICKU(1)
  Its father is: 0

3 FORKS is accepting
    Its entries are:
          PUTDO(0*)  PICKU(1)
  Its father is: 0

4 FORKS is accepting
    Its entries are:
          PUTDO(0*)  PICKU(1)
  Its father is: 0

5 FORKS is accepting
    Its entries are:
          PUTDO(0*)  PICKU(1)
  Its father is: 0

6 FORKS is accepting
    Its entries are:
          PUTDO(0*)  PICKU(1)
  Its father is: 0

7 SOCRA is calling task number 3 at entry PICKU
    Its entries are:
          <NONE>
  Its father is: 0
```

```
8 PLATO is calling task number 4 at entry PICKU
    Its entries are:
          <NONE>
  Its father is: 0

9 ARIST is calling task number 5 at entry PICKU
    Its entries are:
          <NONE>
  Its father is: 0

10 MARX  is calling task number 6 at entry PICKU
    Its entries are:
          <NONE>
  Its father is: 0

-- RUSSELL will be the philosopher task receiving the monitor warning.

11 RUSSE is calling task number 2 at entry PICKU
    Its entries are:
          <NONE>
  Its father is: 0

**MON**  end of dead state description.

**TRC** call of monitor entry CALLING.

  The consumer is 11[RUSSE]. The server is 6[FORKS]. The entry is [PUTDO].

-- This indicates that RUSSELL is taking evasive action and is putting down the lefthand fork
-- instead of attempting to pick up the righthand fork. Note that the monitor call to UNBLOCK
-- indicating evasive action, is not traced, but must already have been called so that the
-- monitor's "picture" is correct.

**TRC** call of monitor entry START_RENDEZVOUS.
  The consumer is 11[RUSSE]. The server is 6[FORKS]. The entry is [PUTDO].

**TRC** call of monitor entry END_RENDEZVOUS.
  The consumer is 11[RUSSE]. The server is 6[FORKS].

-- RUSSELL has now put down his left fork.

**TRC** call of monitor entry CALLING.
  The consumer is 11[RUSSE]. The server is 6[FORKS]. The entry is [PICKU].

-- RUSSELL now attempts to pickup the lefthand fork again!
-- However he will be behind MARX on the entry queue.

**TRC** call of monitor entry ACCEPTING.
  The server is 6[FORKS]. The entry is[PICKU].

-- A FORK, task 6, is the only unblocked task.

**TRC** call of monitor entry START_RENDEZVOUS.
  The consumer is 10[MARX ]. The server is 6[FORKS]. The entry is [PICKU].
```

```
-- Now MARX can pickup his righthand fork, which was RUSSELL's lefthand fork.

**TRC** call of monitor entry END_RENDEZVOUS.
  The consumer is 10[MARX ]. The server is 6[FORKS].

**TRC** call of monitor entry ACCEPTING.
  The server is 6[FORKS]. The entry is [PUTDO].

**TRC** call of monitor entry CALLING.
  The consumer is 10[MARX]. The server is 5[FORKS]. The entry is [PUTDO].

-- Now MARX is finished eating and prepares to put down his forks.

**TRC** call of monitor entry START_RENDEZVOUS.
  The consumer is 10[MARX]. The server is 5[FORKS]. The entry is [PUTDO].

**TRC** call of monitor entry END_RENDEZVOUS.
  The consumer is 10[MARX]. The server is 5[FORKS].

**TRC** call of monitor entry CALLING.
  The consumer is 10[MARX]. The server is 6[FORKS]. The entry is [PUTDO].

**TRC** call of monitor entry ACCEPTING.
  The server is 5[FORKS]. The entry is [PICKU].

**TRC** call of monitor entry START_RENDEZVOUS.
  The consumer is 10[MARX]. The server is 6[FORKS]. The entry is [PUTDO].

**TRC** call of monitor entry END_RENDEZVOUS.
  The consumer is 10[MARX]. The server is 6[FORKS].

**TRC** call of monitor entry CALLING.
  The consumer is 10[MARX]. The server is 1[TABLE]. The entry is [GETUP].

**TRC** call of monitor entry ACCEPTING.
  The server is 6[FORKS]. The entry is [PICKU].

**TRC** call of monitor entry START_RENDEZVOUS.
  The consumer is 9[ARIST]. The server is 5[FORKS]. The entry is [PICKU].

-- ARISTOTLE gets his righthand fork and starts eating.

**TRC** call of monitor entry START_RENDEZVOUS.
  The consumer is 10[MARX]. The server is 1[TABLE]. The entry is [GETUP].

-- Now MARX has left the table.
```

The trace output continues on indefinitely.

7. CONCLUSIONS AND FURTHER RESEARCH

Implementation of this experimental deadness monitor required half a man-year effort after the basic principles had been formulated. During implementation, many of our initial ideas were extended or modified in order to detect different kinds of errors, to account for complexities in the Ada language and its tasking semantics, that had been overlooked initially, and to provide useful diagnostics. An initial implementation as part of a run-time task supervisor package would have been a formidable project.

Experiments with the monitor indicate that these run-time techniques are indeed practical. A wide class of common tasking errors are detected and the diagnostic descriptions are useful. The monitor can be used simply as a debugging aid or it can be integrated permanently into a tasking system to support evasive action and reconfiguration of threads of control.

For use in debugging, it was easy to implement a version of the monitor with interactive facilities. Using this monitor, a programmer can "single step" through his program's task interactions. He can interactively request snapshots of the monitor's current picture, in addition to controlling output tracing the task rendezvous. This version, together with the present preprocessor, could be reimplemented to production quality standards; providing a useful additional component to Ada programming environments.

Software reconfiguration (e.g. evasive action) provides an important alternative to expensive proofs of correctness for the construction of highly reliable tasking systems. Software reconfiguration is likely to be useful for purposes other than avoiding deadness errors, such as recovering from hardware failures and efficient run-time utilization of resources. For these applications the run-time overhead of monitoring should be studied. In the single CPU case, the overhead seems to be linear in the number of task interactions in most cases. However mathematical analysis is very much an open question especially in the multiple CPU case.

Integration with run-time supervisors (also written in Ada) at some future time is clearly possible and not difficult.

There are a number of areas where this work needs to be extended:

1. **Detection of a wider range of deadness errors.** Specifically, in future monitors these should include errors whereby some proper subset of tasks in a system becomes dead, and errors due to operations on shared global variables. (This last case will involve formal annotations recognizable by the monitor preprocessor.)

2. **Improvement of diagnostic descriptions,** especially to pinpoint the source of errors in systems of dynamically allocated tasks. For example, the monitor currently describes a dynamically allocated task in terms of the accessed type. It would be far better to give the proper context when describing such tasks. For example, a binary tree of tasks could be described as: `ROOT`, `ROOT.LEFT`, `ROOT.RIGHT`, `ROOT.LEFT.LEFT`, etc.

3. **Improved user interface.** Currently the programmer has no way to formally annotate the intended task interactions. Consequently, the monitor detects only those deadness errors that may be recognized on the basis of the syntax and semantics of Ada itself; no

knowledge about the program is used. Formal annotations at the task specification level, (e.g. path expressions) for describing which tasks are able to interact, and the order of their interactions, should be developed to enable the monitor to detect a wider class of errors.

The debugging facilities of the monitor should be expanded to include command scheduling, where the programmer controls the order in which tasking statements are executed. Since deadness errors are often difficult to reproduce (especially on loosely coupled processors), this facility is expected to be very useful. Perhaps log files may be fed directly into future monitors, allowing the events leading to a deadness error to be replaced.

4. **Efficient run-time monitoring.** The monitor system outlined here executes as a single task. This task may become a bottleneck when monitored programs are run on multi-processor systems. A distributed monitor design, where the monitor consists of multiple tasks, could alleviate the bottleneck. However, a distributed monitor would have more overhead due to inter-monitor communication and redundancy of data. These tradeoffs need to be examined further. Distributed monitor designs are currently under development.

8. REFERENCES

[1] Ichbiah, J.D., Krieg-Brueckner, B., Wichmann, B.A., Ledgard, H.F., Heliard, J.C., Abrial, J.R., Barnes, J.P.G., Woodger, M., Roubine, O., Hilfinger, P.N., & Firth, R.
The Ada Programming Language Reference Manual.
US Department of Defense, US Government Printing Office, 1983.
ANSI/MILSTD 1815A document.

[2] German, S.M., Helmbold, D.P., & Luckham, D.C.
Monitoring for Deadlocks in Ada Tasking.
In Gargaro, A.B. (editor), *Proceedings of the AdaTec Conference on Ada*, pages 10-25. ACM, Arlington, Virginia, October, 1982.

[3] Li, W.
An Operational Semantics of Multitasking and Exception Handling in Ada.
In Gargaro, A.B. (editor), *Proceedings of the AdaTec Conference on Ada*, pages 138-151. ACM, Arlington, Virginia, October, 1982.

[4] Luckham, D.C., Larsen, H.J., Stevenson, D.R., & von Henke, F.
ADAM — An Ada based Language for Multi-processing.
Technical Report Program Verification Group Report PVG-20, CSD Report STAN-CS-81-867, Stanford University, July, 1981.
Revised 1983, CSL-TR-83-240, to appear in Software Practice & Experience.

[5] Taylor, R.
Complexity of Analyzing the Synchronization Structure of Concurrent Programs.
Information and Computer Sciences Report, University of California, Irvine, August, 1982.

[6] Taylor, R.N.
Analysis on Concurrent Software by Cooperative Application of Static and Dynamic Techniques.
In Hans-Ludwig Hausen (editor), *Proceedings of the Symposium on Software Validation.* North-Holland Publishing, Darmstadt, F.R.G., September 26 - 30, 1983.


DAT
FILM